VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozpoznávání kontextů v TIL
Context Recognition in TIL

2016 **Bc. Michal Fait**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student:	**Bc. Michal Fait**
Studijní program:	N2647 Informační a komunikační technologie
Studijní obor:	2612T025 Informatika a výpočetní technika
Téma:	Rozpoznávání kontextů v TIL Context Recognition in TIL
Jazyk vypracování:	čeština

Zásady pro vypracování:

Cílem práce je naprogramovat v jazyce Prolog systém pro rozpoznávání výskytů ve třech kontextech (extenzionálním, intenzionálním a hyperintenzionálním) v Transparentní intensionální logice.

Práce bude obsahovat:
1. Souhrn teorie TIL.
2. Seznámení s problematikou rozpoznávání kontextů.
3. Implementace v jazyce Prolog - rozpoznávání výskytů.

Práce bude zabudována do většího celku inferenčního stroje pro TIL.

Seznam doporučené odborné literatury:

[1] Duží M., Jespersen B. and Materna P. (2010): Procedural Semantics for Hyperintensional Logic. Foundations and Applications of Trasnsparent Intensional Logic. First edition. Berlin: Springer, series Logic, Epistemology, and the Unity of Science, vol. 17, ISBN 978-90-481-8811-6.
[2] Duží M., Materna P. (2012): TIL jako procedurální logika (průvodce zvídavého čtenáře Transparentní intensionální logikou). Aleph Bratislava 2012, ISBN 978-80-89491-08-7

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Marek Menšík, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 29.04.2016

VYSOKÁ ŠKOLA BÁŇSKÁ - TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne: *28.4.2016*

.....
podpis studenta

Poděkování

Rád bych poděkoval Mgr. Marku Menšíkovi, Ph.D. za vedení mé diplomové práce. Dále bych rád poděkoval doc. RNDr. Marii Duží, CSc. za odbornou pomoc a konzultace.

Abstrakt

Cílem této diplomové práce je implementace systému rozpoznávání tří druhů kontextu, a to *extensionálního*, *intensionálního* a *hyperintensionálního* kontextu, ve kterém se může vyskytovat daná konstrukce systému Transparentní intensionální logiky (*TIL*). Implementace pro komputační variantu systému TIL, tj. funkcionální jazyk *TIL-Script*, je provedena v programovacím jazyce Prolog. Rozpoznání kontextu je základním předpokladem pro vývoj inferenčního stroje TIL-Script, neboť umožňuje správnou aplikaci všech extensionálních logických pravidel odvozování, a to v každém kontextu. V práci jsou nejprve představeny teoretické základy systému TIL, které slouží jako specifikace pro implementaci, a jazyk TIL-Script. Následuje popis lexikální a syntaktické analýzy jazyka TIL-Script a převod do jazyka Prolog. Vlastní algoritmus vychází z báze konstrukcí zapsaných ve formě klauzulí jazyka Prolog. Provádí typovou kontrolu konstrukcí a rozpoznání kontextu jejich výskytu. Výstupem je pak derivační strom konstrukce zapsaný v jazyce XML.

Klíčová slova Transparentní intensionální logika, TIL, TIL-Script, tři druhy kontextu, hyperintensionalita, algoritmus rozpoznání kontextu, Prolog.

Abstract

The goal of this diploma thesis has been implementation of the algorithm for recognising three kinds of context in which constructions of Transparent Intensional Logic (*TIL*) can occur, to wit *extensional*, *intensional* or *hyperintensional* occurrence. The algorithm has been implemented in the Prolog programming language and realized for the computational variant of TIL, the *TIL-Script* functional programming language. Context recognition is the fundamental necessary condition for the development of the TIL-Script inference machine, because it makes it possible to correctly apply all the extensional logical rules of inference in any context. The first part of the thesis deals with theoretical foundations of TIL which in turn serve as the specification of the Prolog implementation. Here we also describe the TIL-Script language. In the second part we introduce the results of lexical and syntactic analysis of TIL-Script constructions and their transformation into Prolog knowledge base. The algorithm for context recognition is introduced in the third part. It operates on the base of constructions specified in the form of Prolog clauses, realizes their type-theoretical control and recognition of the context in which a given construction occurs. As a result the algorithm produces the derivation tree of each construction specified in the XML language.

Key words Transparent intensional logic, TIL, TIL-Script, three kinds of context, hyperintensionality, algorithm of context recognition, Prolog.

Obsah

Seznam ilustrací a tabulek

1. Úvod

V současné době dochází k poměrně bouřlivému rozvoji systémů pro automatické dokazování teorémů, které jsou známy jako systémy HOL.[1] HOL je akronym pro „higher order logic“, tedy logiky vyšších řádů. Tyto logiky jsou většinou různé verze typovaného lambda kalkulu založeného na jednoduché hierarchii typů. Typickým rysem těchto kalkulů je to, že umožňují operovat jak v *extensionálním* kontextu, ve kterém je objektem predikace *hodnota* funkce označené daným lambda termem, tak i v kontextu *intensionálním*, kde je objektem predikace celá *funkce*.

Je zde však další aplikační oblast, která nabývá stále většího významu, a tou je *zpracování přirozeného jazyka*. V dnešní době jsou k dispozici obrovské korpusy textových dat, které je nutno zpracovávat, analyzovat, a budovat pro ně inteligentní dotazovací systémy, které budou schopny odvozovat implicitní komputační znalosti z těchto velkých bází explicitních znalostí. Pro tyto systémy však potřebujeme aplikovat nejen intensionální, ale i *hyperintensionální* logiku, neboť přirozený jazyk je natolik bohatý, že dovede vyjádřit i takový kontext, ve kterém je vlastní význam jiného podvýrazu objektem predikace. V těchto kontextech pak intensionální analýza nestačí, protože zde selhává substituce logicky ekvivalentních výrazů a v dedukčním kalkulu by to vedlo k nekonzistencím.

Transparentní intensionální logika (TIL) je z formálního hlediska *hyperintensionální*, parciální, typovaný lambda kalkul s *procedurální* sémantikou. Významem λ-termů v TIL není funkce označená daným termem, nýbrž *procedura*, jejímž výstupem je označená funkce. Tedy oproti běžným lambda kalkulům založeným na jednoduché teorii typů, je kalkul TIL založen na rozvětvené hierarchii typů, což dále umožňuje operovat nejen v extensionálním a intensionálním kontextu, ale i v kontextu hyperintensionálním, kde je objektem predikace *procedura* produkující danou funkci. Proto je TIL ideálním nástrojem pro aplikace, které vyžadují práci v hyperintensionláním kontextu, zejména pak pro zpracování přirozeného jazyka.

Za tím účelem je budována v rámci projektu GA15-13277S, „Hyperintensionální logika pro analýzu přirozeného jazyka“, komputační varianta systému TIL, což je funkcionální programovací jazyk *TIL-Script.* Aby mohl být realizován inferenční stroj pro tento jazyk, je nutno nejprve implementovat algoritmus pro rozpoznávání tří druhů kontextu, ve kterém se mohou jednotlivé konstrukce TIL vyskytovat, a to kontextu extensionálního, intensionálního a hyperintensionálního.
Cílem této práce byla právě implementace tohoto algoritmu. Implementace je provedena v jazyce Prolog, a to pro komputační variantu TIL, jazyk TIL-Script. Hlavním výsledkem je pak derivační strom konstrukcí TIL, který je výstupem algoritmu, a ve kterém je pro každou konstrukci provedena její typová kontrola a vyznačen kontext, ve kterém se vyskytuje.

[1] Viz [12].

2. Teoretická část

Tato kapitola představuje krátký úvod do systému Transparentní intenzionální logiky (TIL) a teorie tří druhů kontextu v TIL. Rovněž zde představím komputační variantu systému TIL, což je funkcionální jazyk TIL-Script.

2.1 Základní principy TIL

V této kapitole shrnu základní pojmy a principy TIL. Výklad v této kapitole vychází z [1], odkud jsou taktéž citovány definice.

TIL je prostředek pro logickou analýzu přirozeného jazyka (LAPJ). Logickou analýzou je myšleno zavedení formálních prostředků, umožňující odvozování logických důsledků z daných tvrzení. „*Logická analýza přirozeného jazyka (LAPJ) studuje (odhaluje) vztah mezi významem a denotátem, a to za předpokladu, že významy (a tedy i denotáty) jsou již dány jazykovou konvencí.*"[1]. Pro účel logického odvozování již máme různé logiky a pro ně důkazové kalkuly. Například Predikátovou logiku prvního řádu, avšak ta pro účely LAPJ není ideální, protože použití predikátové logiky v přirozeném jazyce může vést k nesmyslným závěrům. Nyní si takový případ ukážeme.

Václav Klaus je presidentem ČR
Jan Sokol se chtěl stát presidentem ČR
_____
Jan Sokol se chtěl stát Václavem Klausem

Tady je něco špatně, i když jsme aplikovali Liebnizovo pravidlo substituce identit, které říká, že ekvivalentní výrazy označující jeden a tentýž objekt by měly být vzájemně substituovatelné.

$$A = B, \Phi(\ldots A \ldots) \Rightarrow \Phi(\ldots B \ldots)$$

Je zřejmé, že se Jan Sokol nechtěl stát Václavem Klausem. V jiných případech však lze toto pravidlo uplatnit správně.

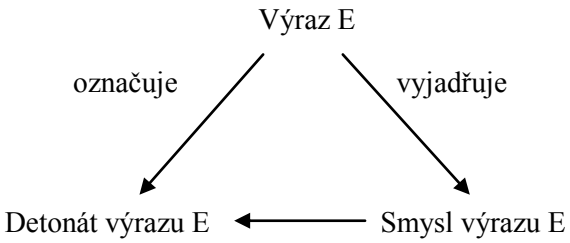
Václav Klaus je presidentem ČR
President ČR je ekonom
_____
Václav Klaus je ekonom

Zde je pravidlo uplatněno a závěr je správný. Předchozí chyba vznikla v důsledku špatné analýzy předpokladů a záměnou různé úrovně abstrakce. V první premise je význam výrazu „president ČR" užit extensionálně, kdežto ve druhé intensionálně. Jestliže se Jan Sokol chtěl stát presidentem ČR, tak měl vztah k tomuto úřadu, ne k jeho (náhodné) hodnotě, tj. k tomu, kdo jej zastává. Pro správnou, přesnější analýzu nelze použít predikátovou logiku, potřebujeme daleko expresivnější logický systém. Predikátová logika 1. řádu nám totiž neumožňuje zachytit postoje mluvčího, analyzovat zmiňování funkcí a vztahů, apod. Pro účely LAPJ se jeví jako ideální právě Transparentní intenzionální logika.

2.1.1 Logické základy TIL

TIL vychází se sémantického schématu, které definoval německý matematik, logik a filozof Gottlob Frege. Schéma výrazu E přiřazuje mimojazykovou entitu, kterou daný výraz označuje, takzvaný detonát. Krom detonátu schéma výrazu přiřazuje také jeho smysl, který výraz vyjadřuje. Frege však nikdy přesně nedefinoval smysl, jen uvedl, že se jedná o jistý „způsob danosti" (reprezentace) detonátu. Různé výrazy pak mohou označovat stejné entity, avšak jiným způsobem.

Fregeho sémantický trojúhelník vypadá následovně:



Frege se zamýšlel nad tím, proč věty typu *a=b* přinášejí jistou informaci, na rozdíl od vět typu *a=a*. Věta typu *a=b* znamená, že výrazy *a* a *b* označují stejný detonát, liší se ve způsobu danosti. Jako konkrétní příklad uvedl Frege rovnost *Jitřenka = Večernice*, oba výrazy označují planetu Venuši, avšak tak činí jiným způsobem. Výraz Jitřenka na planetu Venuši ukazuje jako na „nebeské těleso nejviditelnější na ranní obloze", pro výraz *Večernice* platí obdobná podmínka avšak pro večerní oblohu.

Pro potřeby LAPJ je však Fregeho pojetí sémantiky nevhodné. Jak již bylo zmíněno, LAPJ požaduje, aby detonáty byly jednoznačně dány jazykovou konvencí. Přiřazení Venuše jako detonátu výrazu Jitřenka je nepřípustné, jelikož tento fakt je závislý na empirickém zkoumání. Jazyková konvence, která by přiřazovala Venuši výrazu *Jitřenka*, by musela být vševědoucí. Jako jiný příklad můžeme uvést výraz *největší město Polska*, jehož detonát taky nemůžeme chápat jako Varšavu, protože velikost Varšavy (a ostatních polských měst) je empirický fakt, závislý na zkoumání současného stavu světa.

Otázkou tedy je, jak budeme definovat denotát pro potřeby TIL. Základním typem detonátu v TIL je *intenze.* Nejedná se o objekt, který danou podmínku splňuje, ale o podmínku samotnou. Je to podmínka, kterou daný objekt v konkrétním stavu světa může splňovat. Tento objekt se nazývá *reference.* TIL explikuje takovéto empirické podmínky jako funkce s doménou možných světů, tzv. *intense*. Světem se myslí jistá maximální konzistentní množina empirických faktů. Konkrétní objekt je výrazem označen pouze v případě matematických a logických tvrzení, které jsou platné za všech okolností (detonátem je tzv. *extenze*).

Dále je třeba určit, co budeme v případě TIL považovat za smysl výrazu. Zatím si řekneme, že se jedná o *konstrukci* (denotátu), tj. jistý abstraktní návod, jak získat denotát. Výklad o konstrukcích se nachází v následující kapitole.
Nyní si ještě přiblížíme nejdůležitější rysy TIL:

Respektování principu kompozicionality

Princip kompozicionality říká zhruba toto: *„ Význam (resp. denotát) výrazu E je jednoznačně určen významy (denotáty) smysluplných složek (podvýrazů) výrazu E.“*[1] Kompozicionalita je důležitým předpokladem k tomu, aby bylo možno substituovat výrazy s ekvivalentním významem.

Antikontextualismus

Til je logika transparentní. To znamená, že význam každého výrazu je nezávislý na kontextu. Vraťme se však k příkladu chybného úsudku.

Václav Klaus je presidentem ČR
Jan Sokol se chtěl stát presidentem ČR
———————————————————
Jan Sokol se chtěl stát Václavem Klausem

Podle principu kompozicionality by mělo být tímto způsobem možné substituovat výraz prezident ČR do druhé věty. Přece výraz prezident ČR má v obou větách stejný význam, avšak závěr je očividně nesmyslný. Tento problém si uvědomoval již sám Frege, ten však tento problém vysvětlil *kontextualisticky*. Podle něj v některých „nepřímých“ kontextech je detonátem výrazu jeho smysl, kdežto v přímých kontextech je to objekt, daným výrazem označený. Kontextualismus je však nežádoucí, už jen proto, že nemůžeme rozumět výrazu, dokud neuvedeme kontext. Dále je také absurdní uvažovat, že v prvním předpokladu má výraz prezident ČR jiný význam než v druhém. TIL ukazuje, že význam výrazu je nezávislý na kontextu. V tomto případě je nesmyslnost závěru způsobená tím, že v prvním předpokladu je výraz president ČR *užit* tak, že objektem o kterém se něco vypovídá, je konkrétní individuum zastávající úřad presidenta ČR. V druhém se vypovídá o úřadu prezidenta, tj. individuové roli.

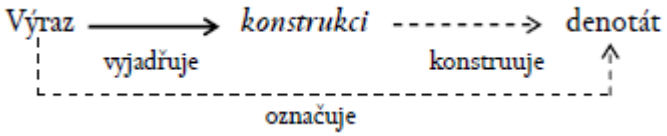
Antiaktualismus

TIL odmítá aktualismus, řídí se sémantikou možných světů. Možný svět je chronologie možných empirických faktů. Výrazy označující intenze neukazují na konkrétní individuum, jen na něj náhodně referují v konkrétním světě a čase. Aktualismus pokládá za detonát právě aktuální referenci. Nejzávažnějším důvodem pro zavedení sémantiky možných světů je především odmítání empirické vševědoucnosti. Kdybychom například považovali za detonát výrazu „hlavní město Polska“ Varšavu, znamenalo by to, že stačí znalost jazyka k tomu, abychom určili, které město hraje roli hlavního města Polska.

2.1.2 Sémantické schéma TIL

Výraz ⟶ *konstrukci*
vyjadřuje jako svůj význam

TIL výrazům přiřazuje konstrukce jako jejich významy. Co to vlastně konstrukce je? Konstrukce jsou algoritmicky strukturované abstraktní *procedury*. Algoritmicky strukturované znamená, že podobně jako počítačový program se konstrukce (krom atomických, dále nedělitelných konstrukcí) skládají z podprocedur (v TIL *konstituentů*). Abstraktní proto, že je pouze specifikováno, co dané konstrukce dělají, bez konkrétní implementace (dá se srovnat s abstraktními metodami v programování). Jazyk konstrukcí je typovaná verze lambda kalkulu. Jedná se o jazyk pro práci s funkcemi. Konstrukce po provedení (provedení si můžeme představit jako spuštění programu) podávají jako výstup (říká se, že konstrukce *v-konstruuje*) objekty (extenze, intenze a dokonce i konstrukce). Nyní si stručně popíšeme, druhy konstrukcí. První dva typy konstrukcí jsou atomické, neskládají se již z jiných konstituentů než sebe sama. Jsou to *proměnné* a *Trivializace* ${}^0X$, která objekt *X* zmiňuje a vrací beze změn. Trivializace je prostě pointer na objekt *X*. Dalším typem konstrukce je *Dvojí Provedení* ${}^2X$ (používá se, pokud konstrukce *X* při prvním provedení konstruuje jinou konstrukci, která může být provedena druhým provedením). Poslední dva typy konstrukcí jsou inspirované lambda kalkulem, a to λ-abstrakce (v TIL *Uzávěr*), která vytváří funkci, a aplikace funkce na argument za účelem získání hodnoty funkce (v TIL *Kompozice*).



Toto je rozšířené schéma TIL. Principem logické analýzy je najít konstrukci, kterou daný výraz vyjadřuje a určit, jakého typu je detonát daného výrazu. Aby byla analýza adekvátní, musí konstrukce konstruovat ten objekt, který výraz označuje (detonát). Za jistých okolností konstrukce nemusí konstruovat nic (buď z důvodu chybného typování, nebo neexistence detonátu). Základním typem detonátu je funkce. Jelikož jsou výrazy, které mají význam, ale nemají detonát (například výraz „poslední desetinné místo čísla π"), musíme uvažovat *parciální* funkce. Funkce jsou *typované*, přijímají objekty daného typu, a je přesně určeno, jaký typ objektu je návratová hodnota funkce. Když je hodnota funkce typu α, a typy argumentů po řadě $\beta_1,\ldots,\beta_n$, tak typ této funkce zapisujeme v TIL jako $(\alpha\ \beta_1\ldots\beta_n)$, a chápeme jako jednoznačné zobrazení z kartézského součinu $\beta_1 \times\ldots\times \beta_n$ do typu α, které pro některé argumenty typů β_1 - β_n nemusí být definováno. Tyto typy můžou být jak prvky báze (základní atomické typy), tak i typy složené, funkcionální. Tedy funkce mohou přijímat jako argumenty a vracet taktéž funkce. TIL je kalkul *hypeintenzionální,* některé funkce mohou operovat i na konstrukcích (takové funkce patří mezi takzvané objekty vyšších řádů). Je potřeba ještě zmínit, že TIL vyjadřuje většinu (krom matematických a logických vztahů, ty považuje za platné za všech okolností) denotátů výrazů jako funkce možných světů. Empirické funkce (*intenze*), označující například vlastnosti individuí, nebo vztahy mezi nimi mají tvar funkce, která bere jeden argument, a to možný svět a hodnotou je funkce, která bere jako argument časový okamžik a hodnotou funkce je již daný objekt v konkrétním světě a čase.

2.1.3 Pojmy a definice TIL

V této části popíšeme a nadefinujeme objekty, které jsou v TIL využívány. Ontologie TIL obsahuje entity extenzionální, intenzionální a hyperintenzionální (konstrukce). Každé entitě TIL ontologie je přiřazen *typ*. Definice nekonečné hierarchie typů je induktivní. Nejprve definujeme bázové typy a funkcionální typy řádu jedna, tj. typy objektů, které nejsou konstrukcemi.

Jednotlivé prvky báze jsou atomické typy, tj. ne-funkce (neboli nulární funkce bez argumentů), které využíváme při analýze v TIL.

Pravdivostní hodnoty – pro logickou analýzu budeme potřebovat dvouprvkovou množinu pravdivostních hodnot Pravda Nepravda – {P, N}.

Univerzum – Množina individuí. Individuem se myslí nejen nějaký konkrétní jedinec, může se jednat také o nějaký konkrétní objekt (například Praha, Mount Everest).

Množina možných světů – což jsou jednotlivé chronologie empirických faktů.

Množina časových okamžiků/reálných čísel – množinu časových okamžiků můžeme definovat jako množinu reálných čísel, protože jich je taktéž nekonečně mnoho. Tedy pro reálná čísla používáme stejný typ jako pro časové okamžiky.

Pro bázové typy používáme v TIL následující značení.

- o množina pravdivostních hodnot
- ι množina individuí
- τ množina časových okamžiků (reálných čísel)
- ω množina logicky možných světů

Nyní uvedu definici typů prvního řádu ([1], str. 39).

Definice 1 (*typy řádu 1*) Nechť *B* je *báze*, tj. kolekce vzájemně disjunktních neprázdných množin. Pak:

i) Každý prvek *B* je atomický (elementární) *typ řádu 1 nad B*.

ii) Nechť α, $\beta_1 \ldots \beta_m$ ($m > 0$) jsou typy řádu 1 nad B. Pak kolekce (α $\beta_1 \ldots \beta_m$) všech *m*-árních parciálních funkcí, tj. zobrazení z Kartézského součinu $\beta_1 \times \ldots \times \beta_m$ do α, je molekulární (neboli funkcionální) *typ řádu 1 nad B*.

iii) Nic jiného není *typem řádu 1 nad B* než dle (i) a (ii).

Skutečnost, že objekt *O* patří do typu α značíme zápisem O/α
Například: Tom/ι, 2/τ, +/(τττ).

TIL striktně rozlišuje empirické a matematické výrazy. Empirické výrazy označují *intenze*, kdežto neempirické (tj. logické a matematické výrazy) označují *extenze*.

Intenze jsou objekty typu (βω), tj. funkce z možných světů do typu β, což často bývá chronologie objektů typu α, tedy objekt typu ((α τ) ω), funkce z možných světů a časů do objektů typu α.
Extenze jsou objekty typu α, kde α ≠ (βω), tedy extenze jsou α–objekty, jejichž doménou není množina možných světů. Extenze nezávisí na daném světě ani čase, jelikož TIL považuje matematické výrazy (taktéž výrazy matematické logiky apod.) za platné ve všech "světočasech".

Intenzionální objekty typu ((α τ) ω) zapisujeme často zkráceně jako $\alpha_{\tau\omega}$

Typy intenzionálních objektů prvního řádu (neobsahující konstrukce):

- *Individuové úřady* – například papež, prezident ČR, mistr světa v gymnastice, označují objekty typu $\iota_{\tau\omega}$, takzvané individuové úřady nebo role.

- *Vlastnosti individuí* – například plešatý, student (přesněji být student), mít rád svou ženu, jsou objekty typu $(o\iota)_{\tau\omega}$. Pro konkrétní stav světa ⟨*w, t*⟩ získáme funkci typu (οι) která nám pro dané individuum vrátí pravdivostní hodnotu dle toho, zda individuum danou vlastnost má nebo ne.

- *Vztahy mezi individui* – rodinné vztahy, mít rád (kdo, koho) - binární relace $(o\iota\iota)_{\tau\omega}$

Typy extenzionálních objektů prvního řádu (neobsahující konstrukce):

- Množina čísel je objekt typu (οτ)

- Binární funkce na reálných číslech (+, -) , jsou objekty typu (τττ)

- Funkce vracející pro danou binární funkci její inverzní funkci je objekt typu ((ττ)(ττ))

- Množina lichých binárních funkcí definovaných na reálných číslech (funkce kde platí rovnost $f(-x) = -f(x)$), je (o(ττ))-objekt

Nyní si můžeme podrobněji popsat, jak vypadají konstrukce. Jazyk konstrukcí je typovaná verze lambda kalkulu, avšak přináší dvě zásadní rozšíření. Prvním z nich je *parcialita,* některé konstrukce mohou mít nedefinovaný výstup, jsou *nevlastní* („improper"). Tato vlastnost je propagována nahoru, tedy molekulární konstrukce, které obsahují mezi svými konstituenty nevlastní konstrukci, jsou taktéž nevlastní. Druhou odlišností je, že jazyk konstrukcí je *hyperintenzionální* kalkul, umožňuje nám pracovat i se samotnou konstrukcí funkce.
Konstrukcí je šest typů, dvě atomické (proměnná a trivializace) a molekulární (kompozice, uzávěr a (dvojí) provedení). Nyní si neformálně popíšeme jednotlivé typy konstrukcí a ukážeme příklady, poté uvedeme formální definici.

Proměnná

Proměnné x, y, z, … jsou atomické konstrukce, konstruují objekty v závislosti na valuaci *v* (valuací je myšleno jednoznačné přiřazení prvků typu α proměnným, například *v(1/x,2/y)* je valuace, která přiřazuje proměnné *x* hodnotu 1, a proměnné *y* hodnotu 2. Každému typu α je přiřazeno spočetně nekonečně mnoho proměnných, které přes něj „rangují", tj. *v*-konstruují objekty typu α.

Trivializace

Druhým typem konstrukce je *Trivializace*, když je aplikována na objekt, vrací ho beze změny. Trivializace objektu X se značí jako ^{0}X. Trivializován může být jakýkoliv objekt, i konstrukce. Používá se jako ukazatel na daný objekt. Dá se chápat taktéž jako prostředek, který zabraňuje implicitnímu provádění konstrukcí.

Příklad:
Konstrukce funkce plus je např. Trivializace $^0+$, konstruuje objekt typu $(\tau\tau\tau)$
Konstrukce čísla 2 je např. ^{02}, konstruuje objekt typu τ

Kompozice

Konstrukce *kompozice* [*F* X_1 ... X_n] je procedura aplikace funkce *f*, *v*-konstruované konstrukcí *F* na *n* argumentů, *v*-konstruované konstrukcemi X_1 ... X_n . Kompozice *v*-konstruuje hodnotu funkce *f* pro dané argumenty. Kompozice selhává při pokusu aplikace funkce na argument, na kterém není definovaná (například dělení nulou), dále v případě chybného typování, nebo při neposkytnutí dostatečného počtu argumentů.

Příklad: aplikace funkce plus na argumenty 1 a 2
$[^0+^01^02]$ - konstruuje objekt typu τ - výsledek operace plus na argumentech 1 a 2 (tj. 3)
Příklad: tvrzení „1 + 2 = 3"
$[^0=[^0+^01^02]^03]$ – konstruuje objekt typu o – výsledek porovnání dvou čísel

Uzávěr

Další konstrukcí je *Uzávěr* [$\lambda x_1 \ldots x_n$ *Y*], který konstruuje anonymní funkci *f*. Pokud konstrukce *Y* konstruuje objekt typu α, a proměnné x_1, …, x_n konstruují po řadě objekty typů β_1, …, β_n, pak je typ funkce konstruované uzávěrem ($\alpha\ \beta_1 \ldots \beta_n$).

Příklad: vytvoření funkce následníka (vnější závorky bývají obvykle vynechávány)
$[\lambda x[^0+\ x\ ^01]]$

Provedení

Provedením konstrukce *X* získáme objekt, který konstruuje. *Provedení* se značí ^1X.
To je však ekvivalentní s *X*, protože TIL konstrukce se implicitně vyskytují v módu provádění. Některé konstrukce mohou být provedeny dvakrát, jelikož konstruují konstrukce, které lze znova provést. Je tedy definováno *Dvojí Provedení* ^2X, které v-konstruuje to, co je *v*-konstruováno *X*.

Příklad: konstrukce $^{20}[^0+^01^02]$ konstruuje číslo 3, jelikož $^0[^0+^01^02]$ konstruuje kompozici $[^0+^01^02]$, ta je podruhé provedena, a konstruuje výsledek operace plus na argumentech 1 a 2.

Nyní uvedeme formální definici konstrukcí.

Definice 2 (*konstrukce*)

i) *Proměnná x* je ***konstrukce***, která konstruuje objekt *O* příslušného typu v závislosti na valuaci *v*; tedy *x v*-konstruuje *O*.

ii) *Trivializace*: Je-li *X* jakýkoli objekt (extenze, intenze nebo i *konstrukce*), pak ^0X je ***konstrukce*** zvaná *Trivializace.* Konstruuje objekt *X* bez jakékoli změny.

iii) *Kompozice* $[X\ Y_1 \ldots Y_m]$ je ***konstrukce***: Je-li *X* konstrukce, která *v-konstruuje* funkci *f* typu $(\alpha\beta_1 \ldots \beta_m)$, a $Y_1, \ldots, Y_m$ *v-konstruují* po řadě objekty $B_1, \ldots, B_m$ typů $\beta_1, \ldots, \beta_m$, pak *Kompozice* $[X\ Y_1 \ldots Y_m]$ *v-konstruuje* hodnotu funkce *f* na argumentech $B_1, \ldots, B_m$ (tj. objekt typu α, pokud *f* má na $\langle B_1, \ldots, B_m \rangle$ hodnotu). Jinak je *Kompozice* $[X\ Y_1 \ldots Y_m]$ *v*-nevlastní, tj. ne (*v*-)*konstruuje* žádný objekt.

iv) *Uzávěr* $[\lambda x_1 \ldots x_m\ Y]$ je ***konstrukce***. Nechť $x_1, x_2, \ldots, x_m$ jsou navzájem různé proměnné, které *v*-konstruují po řadě objekty typu $\beta_1, \ldots, \beta_m$, a nechť *Y* je konstrukce, která *v*-konstruuje α-objekt. Pak $[\lambda x_1 \ldots x_m\ Y]$ *v-konstruuje* funkci $f/(\alpha\beta_1 \ldots \beta_m)$, a to takto: Nechť $v(B_1/x_1,\ldots,B_m/x_m)$ je valuace, která se liší od valuace *v* nanejvýš tím, že přiřazuje objekty $B_1/\beta_1, \ldots, B_m/\beta_m$ proměnným $x_1, \ldots, x_m$. Je-li *Y* $v(B_1/x_1,\ldots,B_m/x_m)$-nevlastní (viz iii), pak funkce *f* není definována na $\langle B_1,\ldots,B_m \rangle$. Jinak je hodnotou funkce *f* na argumentu $\langle B1,\ldots,Bm \rangle$ α-objekt $v(B_1/x_1,\ldots,B_m/x_m)$-konstruovaný *Y*.

v) *Provedení* ^1X je ***konstrukce***, která buď *v*-konstruuje objekt *v*-konstruován konstrukcí *X*, nebo pokud *X* není konstrukce nebo je *v*-nevlastní, je rovněž ^1X *v*-nevlastní, tj. nekonstruuje žádný objekt.

vi) *Dvojí Provedení* ^2X je ***konstrukce***. Tato konstrukce je *v-nevlastní*, pokud *X* není konstrukce, nebo pokud *X* ne *v*-konstruuje jinou konstrukci, nebo *v*-konstruuje *v*-nevlastní konstrukci. Jinak, jestliže *X v*-konstruuje konstrukci *Y* a *Y v*-konstruuje objekt *Z*, pak ^2X *v*-konstruuje *Z.*

vii) Nic jiného není ***konstrukce*** než dle (i) – (vi).

V případě konstrukcí reprezentující empirické výrazy používáme pro proměnné *v*-konstruující prvky typu ω obvykle názvy $w, w_1, \ldots, w_n$, pro proměnné *v*-konstruující prvky typu τ (především časové okamžiky) značíme názvy $t, t_1, \ldots, t_n$.

V případě dvojice kompozic realizující tzv. *intezionální sestup*, tj. dosazení aktuálního světa a času (za účelem získání aktuální reference) ve formě [[C w]t], používáme zápis typu C_{wt}

V případě logických funkcí a operátorů používáme obvykle infixní zápis bez použití Trivializace. Například místo

$$[^0{>}\ p\ q]$$

píšeme

$$[\,p > q\,].$$

.

Jak již bylo řečeno, jazyk TIL je *hyperintenzionální* lambda kalkul a tedy objekty mohou operovat na konstrukcích samotných. Díky tomu můžeme analyzovat věty typu „Tom si myslí, že 2 + 2 je 5“. Jakého typu je objekt označený výrazem „myslí“? Jistě se nebude jednat o vztah Toma a pravdivostní hodnoty konstruované výrazem „2 + 2 je 5“. Tím objektem je samotná konstrukce, tj. význam výrazu „2 + 2 je 5“. Tedy „myslí“ bude objekt typu $(o\iota *_n)_{\tau\omega}$. Zápisem $*_n$ je určeno, že na daném místě funkce

přijímá jako argument (resp. vrací jako návratový typ) konstrukce řádu *n*. Je teda nutno definovat objekty obsahující konstrukce, objekty vyšších řádů.

Definice 3 (*rozvětvená hierarchie typů nad bází B*)
T_1 (*typy řádu 1*) byly definovány v Definici 1.
C_n (*konstrukce řádu n*)

i) Nechť *x* je proměnná, která *v*-konstruuje objekty typu řádu *n*. Pak *x* je *konstrukce řádu n nad B*.
ii) Nechť X je prvek typu řádu *n*. Pak ^0X, ^1X, ^2X jsou *konstrukce řádu n nad B*.
iii) Nechť $X, X_1, ..., X_m$ (m > 0) jsou konstrukce řádu *n* nad B. Pak $[X\ X_1 ...\ X_m]$ je *konstrukce řádu n nad B*.
iv) Nechť $x_1, ..., x_m, X$ ($m > 0$) jsou konstrukce řádu *n* nad *B*. Pak $[\lambda x_1 ...\ x_m\ X]$ je *konstrukce řádu n nad B*.
v) Nic jiného není *konstrukce řádu n nad B* než to, co je definováno dle C_n (i)-(iv).

T_{n+1} (*typy řádu n + 1*)

Nechť $*_n$ je kolekce všech konstrukcí řádu *n* nad *B*.

i) $*_n$ a každý typ řádu *n* jsou *typy řádu n + 1 nad B*.
ii) Jsou-li $\alpha, \beta_1, \ldots, \beta_m$ ($m > 0$) typy řádu $n + 1$ nad *B*, pak $(\alpha\ \beta_1 \ldots\ \beta_m)$, tj. kolekce parciálních funkcí – viz T_1, bod ii), je *typ řádu n + 1 nad B*.
iii) Nic jiného není typ řádu $n + 1$ nad B než dle T_n+1 (i) a (ii).

Skutečnost, že konstrukce *X* typu $*_n$ *v*-konstruuje objekt typu α zapisujeme jako $X \rightarrow_v \alpha$. Pokud *X* konstruuje objekt nezávisle na valuaci *v* (pokud X neobsahuje žádné proměnné), zapisujeme $X \rightarrow \alpha$.

2.1.4 Metoda analýzy a typová kontrola.

Je třeba si ukázat, jak probíhá analýza jazykových výrazů prostředky TIL. Hlavním úkolem je nalezení konstrukce zakódované daným výrazem. Analýza výrazů přirozeného jazyka se řídí takzvaným Parmenidovým principem, podle kterého žádná z podkonstrukcí konstrukce C, která je analýzou výrazu *V* nekonstruuje objekt, který není ve výrazu *V* zmíněn. Analýza, která takovou podmínku splňuje, se nazývá *přípustná analýza*. To, že je analýza přípustná, však nestačí, analýza musí být také *adekvátní*, tedy musí správně popisovat sémantickou strukturu výrazu. Pro adekvátní analýzu aplikujeme tříkrokovou metodu analýzy [1].

1) *Typová analýza* objektů, o kterých daný výraz *V* mluví, tj. těch objektů, které jsou označeny podvýrazy výrazu *V* se samostatným významem.
2) *Syntéza*, tj. „poskládání“ *konstrukcí* objektů *ad* 1) tak, abychom obdrželi konstrukci objektu označeného výrazem *V*.
3) *Typová kontrola*, tj. kontrolujeme, zda byla syntéza provedena v souladu s typovými pravidly vyplývajícími z definice konstrukcí.

Příklady:

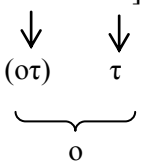
(V_1) „Číslo 5 je sudé číslo“.

Typová analýza: 5/τ, *Sudé*/(οτ), celá věta (*V*1) označuje **P**/o.

Syntéza: $[{}^0Sud\acute{e}\ {}^05]$

Tato Kompozice "říká", že číslo 2 má vlastnost být sudé.

Typová kontrola: $[{}^0Sud\acute{e}\quad {}^05]$



(V_2) „Tom miluje Marii"

Typová analýza: *Tom*, *Marie*/ι, *Miluje* (kdo koho)/$(o\iota)_{\tau\omega}$

Syntéza: Věta říká, že Tom za určitých okolností (dvojice ⟨w, t⟩), miluje Marii. Je to oznamovací věta, která označuje propozici (tj. objekt typu $o_{\tau\omega}$), pravdivostní hodnotu závislou na aktuálním stavu světa. V případě tohoto empirického výrazu si ukážeme jednotlivé kroky syntézy.

Jelikož je objekt *Miluje* intenze, je první třeba provést aplikaci objektu na daný svět, poté na čas. Uvažujeme proměnné $w \rightarrow_v \omega$ a $t \rightarrow_v \tau$. Aplikace vypadá následovně

$$[[{}^0Miluje\ w\ ]\ t\ ].$$

Jak již bylo řečeno, taková dvojice kompozic se zapisuje zkráceně jako

$${}^0Miluje_{wt}\ \rightarrow_v (o\iota).$$

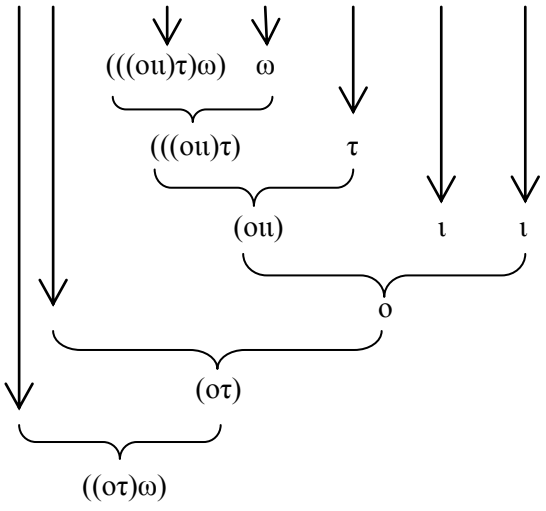
Dále je potřeba aplikovat tento objekt na konstrukce Toma a Marie.

$$[{}^0Miluje_{wt}\ {}^0Tom\ {}^0Marie] \rightarrow_v o.$$

Posledním krokem je třeba abstrahovat proměnné *w*, *t*, abychom vytvořili objekt typu $o_{\tau\omega}$.

$$\lambda w \lambda t\ [{}^0Miluje_{wt}\ {}^0Tom\ {}^0Marie] \rightarrow_v o_{\tau\omega}$$

Typová kontrola: $\lambda w \lambda t[\ [\ [{}^0Miluje \quad w] \quad t] \quad {}^0Tom \quad {}^0Marie]$



Výsledný typ je ((οτ)ω), což je požadovaný typ propozice, zkráceně $o_{\tau\omega}$, označené větou V_2.

2.1.5. Podkonstrukce a konstituenty

Dále si definujeme, co jsou podkonstrukce. Jsou to části dané konstrukce, které jsou také konstrukcemi ([1], str. 61).

Definice 4 (*podkonstrukce*). Nechť *C* je konstrukce. Pak

i) *C* je podkonstrukce *C*.
ii) Je-li *C* tvaru ${}^0X$, ${}^1X$ nebo ${}^2X$ a *X* je konstrukce, pak *X* je podkonstrukce *C*.
iii) Je-li *C* tvaru $[X\, X_1 \ldots X_n]$, pak $X, X_1, \ldots, X_n$ jsou podkonstrukce *C*.
iv) Je-li *C* tvaru $[\lambda x_1 \ldots x_n\, Y]$, pak *Y* je podkonstrukce *C*.
v) Je-li *A* podkonstrukce *B* a *B* je podkonstrukce *C*, pak *A* je podkonstrukce *C*.
vi) Nic jiného není podkonstrukcí *C* než dle (i) – (v).

Je třeba zdůraznit rozdíl mezi pojmy podkonstrukce a konstituent. Konstituenty jsou ty podkonstrukce, které jsou prováděny za účelem získání konstruovaného objektu. Pokud podkonstrukce není konstituentem, jedná se o podkonstrukci *zmíněnou*. Jako příklad můžeme uvést analýzu již dříve zmíněné věty „Tom počítá 5 + 5." jejíž analýza je následující:

Tom/ι, 5/τ, *Počítá*/(οι$*_n$)τω, +/(τττ)
$\lambda w \lambda t[\ {}^0Po\check{c}\acute{i}t\acute{a}_{wt}\ {}^0Tom\ {}^0[\ {}^0+\ {}^05\ {}^05]\ ]$

Konstrukce $[\ {}^0+\ {}^05\ {}^05]$ je zde zmíněna trivializací a použita jako argument pro objekt. Konstrukce není prováděna za účelem poskytnutí konstruovaného objektu. Zmíněná konstrukce může být dokonce nevlastní, avšak její trivializace nikdy neselhává. Všechny podkonstrukce této konstrukce se pak také neřadí mezi konstituenty a jsou zmíněny. To také znamená, že se v rámci celé konstrukce vyskytují *hyperintenzionálně*.

2.1.6 Ekvivalence, *v*-kongurence, procedurální izomorfismus

Dále pro náš výklad v kapitole o rozpoznávání kontextů je třeba si definovat ([1], str. 68), jakým způsobem mohou být konstrukce *ekvivalentní*, popřípadě pouze *v-kongruentní* (slabší případ ekvivalence).

Definice 5 (*ekvivalence a v-kongruence konstrukcí*).
Nechť *C, D*/$*_n \rightarrow \alpha$ jsou konstrukce. Pak *C*, *D* jsou *v-kongruentní*, značíme $C \approx_v D$, jestliže *C* a *D* *v*-konstruují tentýž α-objekt, nebo jsou jak *C* tak *D* *v*-nevlastní. Konstrukce *C*, *D* jsou *ekvivalentní*, značíme $C \approx D$, jestliže *C*, *D* jsou *v*-kongruentní pro všechny valuace *v*, tedy *v*-konstruují tentýž α-objekt nebo jsou obě *v*-nevlastní pro všechny valuace *v* či prostě nezávisle na valuaci.

Když jsou dvě konstrukce *v*-kongurentní, pak to znamená, že konstruují stejný objekt náhodou, pro jednu konkrétní valuaci *v* (jednoduše řečeno když za stejné volné proměnné dosadíme stejné hodnoty pro obě konstrukce). Například konstrukce

$$[^0+\ x\ ^01]\text{ a }[^0-\ ^05\ x]$$

jsou *v(2/x)*-kongruentní, protože pro valuaci *v(2/x)* konstruují číslo 3.

Pokud jsou dvě konstrukce *v*-kongruentní, znamená to, že výrazy, jimž jsou přiřazeny jako jejich významy, jsou *koreferenční*. Takové výrazy sdílí aktuální referenci. Jako příklady koreferenčních výrazů lze uvést „hlavní město ČR“ a „město, kde se nachází Karlův Most“, které jsou koreferenční, protože v aktuálním světě *w* a čase *t* referují ke stejnému objektu, tj. město Praha.

Uvedeme ještě příklad ekvivalentních konstrukcí. Nechť *Power*/(ττ) je funkce mocnina na reálných číslech, a *Multiply*/(τττ) je funkce násobení na reálných číslech. Poté jsou následující konstrukce ^0Power, $\lambda x[\ ^0Multiply\ x\ x]$, ekvivalentní, protože nezávisle na valuaci obě konstruují jednu a tutéž funkci typu (ττ). Pokud jsou konstrukce ekvivalentní, jsou ekvivalentní i jejich výrazy. Jako příklad ekvivalentních výrazů můžeme uvést například „sopka“ a „hora, která chrlí lávu“.

Dále je potřeba říct, co to znamená, že dvě konstrukce jsou *procedurálně izomorfní*. Více o procedurálním isomorfismu viz [5] a [7]. Zde si jen uvedeme neformální objasnění. Pokud jsou dvě konstrukce procedurálně izomorfní, jsou analýzou stejného výrazu nebo více synonymních výrazů se stejným významem, avšak nemusí být zcela totožné. Lišit se můžou například v pojmenování λ-vázaných proměnných, což je *α-ekvivalence*. Dalším způsobem, jak můžeme vytvořit procedurálně izomorfní konstrukci je aplikováním tzv. *η-redukce.*

Zhruba řečeno, *η-redukce* spočívá v aplikaci funkce na argumenty dodané proměnnými a následném abstrahování od hodnot těchto proměnných. Nyní si ukážeme příklad *η-redukce* (*Chytrý*/$(o\iota)_{\tau\omega}$, $x \rightarrow_v \omega$):

$$^0Chytrý$$
$$[\ ^0Chytrý\ x\ ]$$
$$\lambda x\ [\ ^0Chytrý\ x\ ]$$

První a třetí konstrukce jsou procedurálně izomorfní a konstruují objekt typu $(o\iota)_{\tau\omega}$. Obecně, dvě konstrukce C_1 a C_n jsou procedurálně izomorfní, pokud existuje posloupnost konstrukcí C_1, C_2, …, C_n taková, že pro každé dvě po sobě následující konstrukce C_i a C_{i+1} platí, že C_{i+1} vznikla z C_i buď operací přejmenování λ-vázaných proměnných nebo metodou η-transformace.

Možná trochu zvláštní je, že procedurálně izomorfní konstrukce se považují za významy synonymních výrazů. Například konstrukce $^0Košíková$ a $^0Basketbal$ jsou nejen procedurálně izomorfní, ale dokonce identické. Je tomu tak proto, že obě konstrukce konstruují stejný objekt přes jeho název, což je z procedurálního hlediska stejná operace. Nezáleží na tom, jak onen objekt nazveme, Trivializace jednoho a téhož objektu je jedna a tatáž konstrukce, jinými slovy pointer na tento objekt.

To, že dvě konstrukce *C*, *D* jsou procedurálně izomorfní, značíme zápisem $^0C =_* {}^0D$, v našem případě $^{00}Košíková =_* {}^{00}Basketbal$.

2.2 Rozpoznávání kontextů v TIL

V této části se seznámíme s problematikou rozpoznávání kontextů. Určení kontextu je důležitým předpokladem k tomu, abychom určili, jak v TIL správně aplikovat extenzionální pravidla odvozování. To je totiž nutným předpokladem pro budování *extenzionální logiky* hyperintensí, ve které jsou všechna logická pravidla platná nezávisle na kontextu, pouze musí být správně aplikována na objekt toho typu, který je v tom či onom kontextu objektem predikace (argumentem dané funkce).

Nejzákladnější extenzionální pravidla jsou Leibnizovo pravidlo substituce identit a existenční generalizace. Bez správné analýzy a rozpoznání kontextu by dedukce mohla vést k nesmyslným a paradoxním závěrům. Obě extenzionální pravidla si nyní představíme.[2]

A) Substituce identit: Jestliže $a = b$, pak $C(a/x) = C(b/x)$. Můžeme je zapsat rovněž ve formě pravidla takto: $a = b$, $C(a/x) \vdash C(b/x)$.

   Pravidlo nám říká, že výrazy označující jeden a tentýž objekt by měly být vzájemně nahraditelné. Vraťme se k příkladu z kapitoly 2.1. a podívejme se na již známé úsudky.

Václav Klaus je presidentem ČR
President ČR je ekonom
———————————————
Václav Klaus je ekonom

Václav Klaus je presidentem ČR
Jan Sokol se chtěl stát presidentem ČR
———————————————
Jan Sokol se chtěl stát Václavem Klausem

Otázkou je, proč v prvním případě je úsudek platný, zatímco ve druhém případě evidentně není. Je-li Václav Klaus identický s individuem, které zastává úřad presidenta ČR (první premisa), pak by měl být substituovatelný za prezidenta ČR, nebo ne? Jistě, ovšem pouze v takovém kontextu, kdy objektem predikace je to individuum (pokud takové existuje), které zastává úřad prezidenta ČR, tj. náhodná *hodnota* této funkce. Říkáme také, že výskyt výrazu „prezident ČR“ je v první premise *extenzionální.* To je případ prvního úsudku. Je-li však objektem predikace *celý úřad* jakožto *funkce*, pak můžeme substituovat pouze výraz označující stejný úřad, stejnou funkci, což je případ druhého úsudku. Tedy výskyt výrazu „prezident ČR“ je ve druhé premise druhého úsudku *intensionální.* Tak např. pokud jsou výrazy „prezident ČR“ a „hlava státu ČR“ ekvivalentní, tj. označují stejný úřad, pak je následující úsudek platný:

[2] Výklad v této kapitole částečně využívá se svolením autorky dosud nepublikovaný článek M. Duží „Procedurální sémantika TIL“.

Prezident ČR je hlava státu ČR
Jan Sokol se chtěl stát presidentem ČR

Jan Sokol se chtěl stát hlavou státu ČR

B) Existenční generalizace: $C(a/x) \vdash \exists x\, C(x)$.
Toto pravidlo říká, že to, o čem něco vypovídáme, musí existovat.

Papež je plešatý

Papež existuje

V tomto případě se jedná o platný úsudek.

Tom chce být Papežem

Papež existuje

Došli jsme k paradoxu. Tomovo přání rozhodně nemůže být dostatečným předpokladem k tomu, abychom mohli správně usoudit, že Papež existuje. Jistě, Tom si může přát být papežem i v takovém stavu světa, kdy papež neexistuje a Tom si přeje, aby konkláve zvolila právě jeho.

Problém, proč první úsudek je platný a druhý ne, je zcela analogický jako v případě (A). V prvním úsudku je objektem predikace, jemuž je připisována plešatost, to individuum, které zastává úřad papeže, tedy *hodnota* tohoto úřadu. Jedná se o extenzionální výskyt. Ve druhém případě je však objektem predikace *celý úřad*, kterému je přisuzována vlastnost, že Tom jej chce zastávat. Jedná se o intensionální výskyt. Proto můžeme platně odvodit pouze to, že existuje úřad, který chce Tom zastávat, avšak ne to, že tento úřad je obsazen.

2.2.1 Tři druhy kontextu

Základním rysem TIL a ostatních λ-kalkulů je striktní rozlišování mezi funkcí a hodnotou funkce. TIL ještě navíc umožňuje rozlišovat mezi funkcí a zadáním funkce (konstrukcí). TIL tedy rozlišuje tři formy abstrakce. Na základě těchto úrovní abstrakce jsou definovány tři druhy kontextu.

Kontextem se myslí to, jakým způsobem se daná konstrukce vyskytuje v nějaké svojí nadkonstrukci. Může mít výskyt extenzionální, intensionálním nebo hyperintensionální. Rozpoznání těchto kontextů umožňuje definovat takzvanou extenzionální logiku hyperintenzí, tj. kalkul, ve kterém jsou extensionální pravidla platná. Je třeba však pravidla správně aplikovat, tj. vzít v úvahu, objekt jakého typu je argumentem, na který je funkce aplikovaná.

Nyní si můžeme tyto tři způsoby neformálně definovat [1] (formální definice jsou uvedeny dále, v kapitole 3.2.3).

Konstrukce *C* se vyskytuje v *D hyperintenzionálně*, pokud je celá konstrukce *C* argumentem nějaké funkce konstruované v rámci *D*. Poté se všechny podkonstrukce *C* vyskytují v *D* také *hyperintenzionálně* (vyšší, hyperintenzionální kontext je dominantní nad nižším intensionálním nebo extensionálním). Můžeme tedy říct, že *C* není v *D* užita (pro konstruování funkce), nýbrž je zmíněna jako objekt / argument jiné funkce.

Konstrukce *C* se vyskytuje v *D intenzionálně*, pokud to není výskyt v *hyperintenzionálním kontextu D*, a konstrukce *C* konstruuje nějakou funkci *f*, která je jakožto celá funkce argumentem jiné funkce konstruované v rámci *D*. V tomto případě mají všechny konstituenty *C* výskyt taktéž *intenzionální*, jedná se o *intenzionální kontext.*

Konstrukce *C* se vyskytuje v *D extenzionálně,* pokud není tento výskyt v *hyperintenzionálním* nebo *intenzionálním* kontextu *D*, a *C* konstruuje nejen funkci *f*, ale také hodnotu funkce *f* na nějakém argumentu *a*. Tato hodnota je pak je pak argumentem jiné funkce konstruované v rámci *D*. V tomto případě se jedná o *extenzionální kontext*.

Tedy hyperintenzionální kontext je dominantní nad intenzionálním a intenzionální dominuje nad extenzionálním. Znamená to, že když je konstrukce použita extenzionálně a tento výskyt je v rámci intenzionálního kontextu, platí vyšší kontext.

Příklady:

$[^0+^0 1\,^0 2]$

V této konstrukci se vyskytuje extenzionálně konstrukce $^0+ \rightarrow (\tau\tau\tau)$, která je aplikována na argumenty 1 a 2. Intenzionálně se vyskytují ostatní konstituenty, tj. $^0 1 \rightarrow \tau$, $^0 2 \rightarrow \tau$ a $[^0+^0 1\,^0 2] \rightarrow \tau$, které konstruují čísla, tj. nulární funkce bez argumentů, které již nemohou být aplikovány na žádný argument.

$\lambda w \lambda t[^0 Počítá_{wt}\, ^0 Tom\, ^0[^0+^0 1\,^0 2]]$

V této konstrukci se nachází konstrukce $[^0+^0 1\,^0 2]$ hyperintenzionálně, konstrukce je zmíněna Trivializací jako objekt, se kterým pracuje funkce Počítá. Taktéž všechny výskyty podkonstrukcí této Kompozice jsou hyperintenzionální. Konstrukce $^0 Tom \rightarrow \iota$ a celá konstrukce $[^0 Počítá_{wt}\, ^0 Tom\, ^0[^0+^0 1\,^0 2]\,]$ se vyskytují intenzionálně, konstruují funkci, která není aplikována na žádný argument. $^0 Počítá \rightarrow (o\iota *_n)_{\tau\omega}$ se vyskytuje extenzionálně v konstrukci $[^0 Počítá_{wt}\, ^0 Tom\, ^0[^0+^0 1\,^0 2]]$, funkce je užita ke konstruování hodnoty své funkce na argumentech. Jelikož se jedná o empirický výraz, pro které platí, že pokud výskyt konstrukce C v konstrukci $\lambda w \lambda t[\ldots C \ldots]$ je stejný jako výskyt v konstrukci $[\ldots C \ldots]$, $^0 Počítá \rightarrow (o\iota *_n)_{\tau\omega}$ se vyskytuje extenzionálně i v celém uzávěru.[3]

2.2.2 Extensionální pravidla pro logiku hyperintensí

Rozlišení tří druhů kontextu nám umožňuje realizovat extensionální logiku hyperintensí, tj kalkul, ve kterém budou platná extenzionální pravidla odvozování za všech okolností, tj. v každém kontextu, jen musí být správně aplikována. V každé kontextu si musíme uvědomit, na jaké úrovni abstrakce operujeme a podle toho určit, jak můžeme dané pravidlo použít. Více informací včetně podrobnější specifikace můžeme najít například v [9], [10], [11].

Nyní uvedeme jednotlivá extenzionální pravidla pro tři úrovně abstrakce.

[3] Jedná se o výjimku, jelikož v jiných případech operace aplikování uzávěru generuje intenzionální kontext, který přebije všechny extenzionální výskyty.

Existenční generalizace

Při uplatňování pravidla existenční generalizace musíme vzít v úvahu, že TIL je logika parciálních funkcí. Pracujeme s nevlastními konstrukcemi, které nemusí konstruovat žádný objekt, pokud funkce *f* není na argumentu *a* definována. Jak již bylo řečeno, procedura aplikace funkce na argument vytváří extenzionální kontext. Při uplatňování pravidla existenční generalizace musíme nejprve zkontrolovat, zda konstituenty dané konstrukce, které se vyskytují extenzionálně, nejsou *v*-nevlastní. Pokud tomu tak je, nelze pravidlo existenční generalizace aplikovat.

Nyní uvedeme tato pravidla schematicky. Nechť $F \rightarrow_v (\beta\alpha)$, $A \rightarrow_v \alpha$, $[F\ A] \rightarrow_v \beta$.

A) Extenzionální kontext

Nechť je výskyt $[F\ A]$ v konstrukci $[\ldots [F\ A] \ldots]/*_n \rightarrow_v o$ extenzionální a tato konstrukce není *v*-nevlastní. Pak následující pravidlo zachovává pravdivost:

$$[\ldots [F\ A] \ldots] \vdash [^0\exists \lambda x\ [\ldots [F\ x]\ \ldots]]; \quad x/*_n \rightarrow_v \alpha$$

Příklad:

„Michal je plešatý“ ⊨„Existuje někdo plešatý“

$\lambda w \lambda t [^0 Plešatý_{wt}\ ^0 Michal] \models \lambda w \lambda t \exists x [^0 Plešatý_{wt}\ x]$;

Typy: *Plešatý*/$(o\iota)_{\tau\omega}$, *Michal*/ι.

B) Intenzionální kontext

Nechť je výskyt $[F\ A]$ v konstrukci $[\ldots [F\ A] \ldots]/*_n \rightarrow_v o$ intenzionální a tato konstrukce není *v*-nevlastní. Pak následující pravidlo zachovává pravdivost:

$$[\ldots [F\ A] \ldots] \vdash [^0\exists \lambda f\ [\ldots [f\ a]\ \ldots]];\ f/*_n \rightarrow_v (\beta\alpha)$$

Příklad: *„Michal si myslí, že nejvyšší představitel USA je Afroameričan “ ⊨ „Existuje úřad takový, že si Michal myslí, že ten, kdo jej zastává je Afroameričan“.*

$$\lambda w \lambda t [^0 Myslí_{wt}\ ^0 Michal\ \lambda w \lambda t [^0 Afroameričan_{wt}\ ^0 NejPrestavitelUSA_{wt}\ ]\ ]$$
$$\models \lambda w \lambda t\ \exists f [^0 Myslí_{wt}\ ^0 Michal\ \lambda w \lambda t [^0 Afroameričan_{wt}\ f\ ]\ ]$$

Typy: *Myslí*$(o\iota o_{\tau\omega})_{\tau\omega}$ – vztah individua k propozici, *NejPrestavitelUSA*/$\iota_{\tau\omega}$ - individuová role, *Michal*/ι, $f \rightarrow_v \iota_{\tau\omega}$.

C) Hyperintenzionální kontext

Nechť je výskyt $[F\ A]$ v konstrukci $[\ldots [F\ A] \ldots]/*_n \rightarrow_v o$ hyperintenzionální a tato konstrukce není *v*-nevlastní. Pak následující pravidlo zachovává pravdivost:

$$[\ldots [F\ A] \ldots] \vdash\ [^0\exists c\ [\ldots\ c \ldots]];\quad c/{*_{n+1}}\ \rightarrow_v\ {*_n}\ ;\ {}^2c\ \rightarrow_v \beta$$

Příklad: *„Michal počítá 1 + 2“* ⊨ *„Michal něco počítá“*

$\lambda w\lambda t[^0Počítá_{wt}\ ^0Michal\ ^0[^0+\ ^01\ ^02]\ ] \vDash \lambda w\lambda t\exists c[^0Počítá_{wt}\ ^0Michal\ c\ ]$
Typy *Michal*/ι, *1,2*/τ, *Počítá*/(οι$*_n$)$_{τω}$, +/(τττ).

Substituce identit

a) ***V extenzionálním kontextu*** je platná substituce *v-kongruentních* konstrukcí.
Příklad:

„President USA je Barrack Obama“
„President USA je Afroameričan“
———————————
„Barrack Obama je Afroameričan“

Z analýzy je platnost úsudku zřejmá:
$\lambda w\lambda t[^0President_{wt}\ ^0USA\ ]_{wt} \approx_v\ ^0Barrack$ předpoklad
$[^0Afroameričan_{wt}\ \lambda w\lambda t[^0President_{wt}\ ^0USA\ ]_{wt}]$ předpoklad
———————————
$[^0Afroameričan_{wt}\ ^0Barrack\ ]$ substituce identit

Typy: *Usa*/ι, *Barrack(Obama)*/ι, *President*/(ιι)$_{τω}$, *Afroameričan*(οι)$_{τω}$, $\approx_v$/(οιι).

b) V intenzionálním kontextu je platná substituce *ekvivalentních* konstrukcí.
Příklad:

„Barrack Obama se chce stát Presidentem USA “
„Prezident USA je nejvyšší představitel USA“
———————————
„Barrack Obama se chce stát nejvyšším představitelem USA“

Úsudek je platný, neboť druhá premisa má být čtena intensionálně, tj. tak, že zadává identitu úřadu označeného výrazy „prezident USA“ a „nejvyšší představitel USA“.

Nechť ≈ je relace typu (οι$_{τω}$ι$_{τω}$), tj. identita individuového úřadu, další typy jsou:
Chce_se_stat/(οιι$_{τω}$)$_{τω}$, *President*/(ιι)$_{τω}$, *Nej_Pred*/(ιι)$_{τω}$, *USA*/ι, $\lambda w\lambda t\ [^0President_{wt}\ ^0USA] \rightarrow \iota_{τω}$, $\lambda w\lambda t\ [^0Nej\_Pred_{wt}\ ^0USA] \rightarrow \iota_{τω}$

Pak dostáváme tyto konstrukce:

$\lambda w\lambda t\ [^0Chce\_se\_stat_{wt}\ ^0Barrack\ \lambda w\lambda t\ [\ ^0President_{wt}\ ^0USA]]$
$\lambda w\lambda t\ [\ ^0President_{wt}\ ^0USA] \approx \lambda w\lambda t\ [\ ^0Nej\_Pred_{wt}\ ^0USA]$
———————————
$\lambda w\lambda t\ [^0Chce\_se\_stat_{wt}\ ^0Barrack\ \lambda w\lambda t\ [\ ^0Nej\_Pred_{wt}\ ^0USA]\ ]$

c) V hyperintenzionálním kontextu je platná substituce *procedurálně izomorfních* konstrukcí.

Přiklad:
„Tom řeší rovnici $x + 5 = 10$“
———————————
„Tom řeší rovnici $y + 5 = 10$“

Důkaz:

$[{}^0Resi_{wt}\ {}^0Tom\ {}^0[\lambda x[[{}^0{+}x\ {}^05] = {}^010]]\ ]$	předpoklad
${}^0[\lambda x[[{}^0{+}\ x\ {}^05] = {}^010]] =_*\ {}^0[\lambda y[[{}^0{+}\ y\ {}^05] = {}^010]]$	předpoklad
$[{}^0Resi_{wt}\ {}^0Tom\ {}^0[\lambda y[[{}^0{+}y\ {}^05] = {}^010]]\ ]$	Liebniz, 2

Typy: Tom/ι, $Resi/(o\iota*_n)_{\tau\omega}$, $x,y \rightarrow_v \tau$.

2.3 TIL – Script

TIL-Script je komputační varianta Transparentní intenzionální logiky. Oproti jazyku, ve kterém jsou zapisovány konstrukce TIL neobsahuje syntax jazyka TIL-Script indexy, řecké symboly a podobně. Následující kapitola stručně popisuje nejnovější verzi jazyka TIL-Script, původně navrženému v [3]. Gramatika Til-Skriptu je přiložena jako příloha A.

2.3.1 Datové typy

Tabulka 1 ukazuje, jak jsou tradiční TIL typy vyjádřeny v TIL-Scriptu.
Namísto řeckých písmen jsou typy vyjádřeny přímo jejich názvem. Výrazným rozdílem je, že TIL-Script má typy pro čísla a pro vyjádření časového okamžiku zvlášť. Pro čísla jsou typy dva – celá a reálná čísla. Dalším novým typem je String. Posledním rozdílem je, že u konstrukcí neurčujeme její řád (uvažujeme obecně konstrukci řádu *n*).

TIL typ	TIL-Script typ	Popis
o	Bool	Množina pravdivostních hodnot
ι	Indiv	Množina individuí (universum diskursu)
τ	Time	Množina časů
ω	World	Množina možných světů
-	Int	Množina čísel typu Integer
τ	Real	Množina čísel typu Real
α	Any	Nespecifikovaný typ
$*_n$	*	Množina konstrukcí řádu n

Tabulka 1 - typy v jazyce TIL-Script

Typ zápisu parciální funkce je stejný, v závorkách uvádíme jako první návratový typ, poté typy argumentů. Pro zápis často používaných funkcí typu $\alpha_{\tau\omega}$ se v TIL-Scriptu používá zápis α@tw.

TIL-Script také umožňuje pracovat se seznamy. Seznam definujeme klíčovým slovem list a datovým typem, například List (Real) – seznam reálných čísel.

2.3.2 Definice entit a proměnných

TIL-Script umožňuje definovat typ notační zkratky pro zápis typu, abychom usnadnili opakované použití stejného typu. Tyto definice se uvádí klíčovým slovem TypeDef. Entity se definují běžně jako v TIL, velkým písmenem název a za lomítkem datový typ, do kterého daná entita patří (tady může být použit název typu definovaného v TypeDef). Proměnná a její typ, přes který proměnná ranguje, se definují podobně jako entity. Název proměnné musí začínat malým písmenem a datový typ od názvu oddělují znaky „->“ reprezentující symbol $\rightarrow_v$. Takto definovaná proměnná se považuje za globální, na rozdíl od λ-proměnných které se považují za lokální, platné jen v konkrétním uzávěru.

Ukázka definicí typu a proměnných v TIL-Scriptu:

```
person/Indiv.
PresidentOf/(Bool Indiv Indiv)@tw.
i,j->Int.
TypeDef Tprop := (Bool Indiv)@tw.
TypeDef Toffice := Indiv@tw.
White/Tprop.
Student/Tprop.
PresidentOfTheCR/Toffice.
```

2.3.3 Konstrukce

Proměnné

Zápis proměnné se v TIL-Scriptu od zápisu v TIL nijak neliší. Název proměnné musí začínat malým písmenem a obsahuje pouze alfanumerické znaky a znak podtržítka.

Trivializace

Trivializace se v TIL-Scriptu značí znakem jednoduché uvozovky – „’“.

Kompozice

V kompozici není žádný rozdíl mezi TIL a TIL-Scriptem. Pro dvojici kompozic vyjadřující intenzionální sestup používáme místo zápisu C_{wt} zápis `C@wt`. Proměnné *w*, *t* je možno také indexovat.

Uzávěr

`[\x1:α1, … , xn:αn Y]` je uzávěr v TIL-Scriptu. Syntaxe je podobná jako v TIL. Znak λ je nahrazen zpětným lomítkem – „\“. Dalším rozdílem je, že musí být specifikovány typy lambda proměnných α_1- α_n. Tyto typy jsou uvedeny za dvojtečkou, která následuje po názvu proměnné. Pokud typ není uveden, použije se typ globální proměnné s odpovídajícím názvem.

Provedení

Na rozdíl od TIL se namísto napevno definovaného provedení a dvojího provedení požívá *n*-provedení. Značí se následujícím způsobem - ^n C, n - krát provádíme konstrukci C.

Příklad: vytvoření funkce následníka a následná aplikace na číslo 5.
[[\x:Int [’+ x ’1]] ’5].

3. Implementace

V této kapitole popíšu implementaci systému pro rozpoznávání kontextů v konstrukcích TIL-Script. Program zpracovává TIL-Script soubor a výstupem je soubor XML s uloženými konstrukcemi ve formě XML stromu s jejich výskyty a typem, který konstrukce konstruuje. Tento XML výstup zahrnuje také informace o entitách a globálních proměnných.

Nejprve je však třeba provést analýzu vstupního souboru, která se skládá ze dvou částí. *Lexikální analýza* je analýza znaků a řetězců ve vstupním souboru. *Syntaktická analýza* kontroluje, zda jsou tyto znaky a řetězce poskládány správně podle syntaktických pravidel. Pro tyto účely vzniklo mnoho nástrojů generujících syntaktický a lexikální analyzátor v objektově orientovaných programovacích jazycích. Implementace využívá nástrojů C#Lex a C#Cup[4] pro tvorbu lexikálního a syntaktického analyzátoru v jazyce C#. Během syntaktické analýzy vedle kontroly probíhá i zpracování a ukládání prvků TIL-Scriptu. Syntaktický analyzátor komunikuje s Prologem a po analýze tyto prvky (entity, proměnné a konstrukce) uloží do báze znalostí. Poté spustí několik podprogramů v Prologu. Tyto podprogramy jsou jádrem systému, realizují samotné rozpoznání kontextu včetně typové kontroly a výpisu do XML.

3.1 Zpracování TIL-Scriptu a převod na predikáty jazyka Prolog

V této části si přiblížíme, jak probíhá lexikální a syntaktická analýza jazyka TIL-Script, s následným uložením sestavených struktur do Prologu.

Gramatika TIL-Scriptu je definována zápisem v EBNF formě. EBNF je notace pro zápis bezkontextových gramatik. Tato gramatika umožňuje vyjádřit nepovinné (v hranatých závorkách) a opakující se výrazy (složené závorky) bez mezipravidel, což pro syntaktický analyzátor C#Cup není ideální, jelikož zápis pravidel tohoto typu neumožňuje. Proto bylo potřeba taková pravidla přepsat následujícím způsobem:

Nepovinný výskyt:
Neterminal1 → [Neterminal2]

Po úpravě : Neterminal1 → Neterminal2| ε

Opakování:
Neterminal1 → {Neterminal2}

Po úpravě : Neterminal1 → Neterminal1 Neterminal2 | ε

Problémem je, že nově vzniklá pravidla typu X → ε vytváří takzvanou vypouštějící gramatiku, a pro využití gramatiky v nástroji C#Cup je třeba je eliminovat. Vypouštějící gramatika je nežádoucí, jelikož je ve fázi lexikální analýzy nemožné analyzovat tokeny reprezentující prázdné slovo. Tento problém lze vyřešit aplikováním algoritmu pro vytvoření nevypouštějící gramatiky ([4], str. 159).

[4] http://www.seclab.tuwien.ac.at/projects/cuplex/

Prvně je nutno vyřadit pravidla typu $X \rightarrow \varepsilon$. Dále je potřeba pravidla, která obsahují neterminály, ze kterých jde vygenerovat prázdné slovo, nahradit množinou všech možných pravidel, které vznikly vypuštěním několika z nich. Všechny způsoby, kterými lze z *n* prvků několik prvků vynechat, tvoří takzvanou potenční množinu, která obsahuje všechny podmnožiny těchto *n* prvků, kterých je 2^n. Takže z jednoho pravidla, které obsahuje *n* takových neterminálů, vznikne 2^n (!) pravidel (popřípadě $2^n - 1$, pokud by vypuštění všech najednou vedlo k vytvoření pravidla $X \rightarrow \varepsilon$, které pochopitelně do výsledné množiny nezahrneme).

Tento algoritmus má nevýhodu, že příliš narůstá počet gramatických pravidel, jelikož některá pravidla gramatiky TIL-Scriptu obsahují i čtyři a více výskytů neterminálu *optional whitespace*, ze kterého je možno vygenerovat prázdné slovo. V takovém případě podle zmíněného algoritmu vysoce narostl počet pravidel. Z tohoto důvodu jsou výskyty *whitespace* ignorovány (s výjimkou míst, kde je mezera nutná samozřejmě) během lexikální analýzy a nezahrnovány do gramatických pravidel. V ostatních případech jsou pravidla podle algoritmu upravena, aby počet neterminálů potencionálně generujících prázdné slovo již nebyl tak vysoký.

3.1.1 Lexikální analýza

Úkolem lexikální analýzy je rozdělit vstupní text na tokeny (názvy entit, proměnných, závorky, operátory a ostatní speciální symboly či posloupnosti znaků), které specifikujeme pomocí regulárních výrazů. Tyto tokeny pak vstupují do syntaktické analýzy jako terminály. Nyní si ukážeme specifikační soubor pro C#Lex s definicí regulárních výrazů pro tokeny a s akcemi v kódu, které se při rozpoznání daných tokenů mají provést.

```
using TUVienna.CS_CUP.Runtime;
using System;
namespace Parser;

%%

%cup

WHITESPACE=[ \t\r\n\f]+
UPLNAME=[A-Z]([a-z]|[_]|[A-Z])*
LOWLNAME=[a-z]([a-z]|[_]|[A-Z]|[0-9])*
BASETYPE=Bool|Indiv|Time|String|World|Real|Int|Any
INTNUMBER=[0-9]+
REALNUMBER=[0-9]+(\.[0-9]+)?

%%
"TypeDef"{WHITESPACE}  { return new Symbol(sym.TYPEDEF); }
"ObjectDef"{WHITESPACE}  { return new Symbol(sym.OBJECTDEF); }
"+" { return new Symbol(sym.PLUS); }
"*" { return new Symbol(sym.TIMES); }
"/" { return new Symbol(sym.DIVIDE); }
"-" { return new Symbol(sym.SUB); }
">" { return new Symbol(sym.GREATER); }
"=" { return new Symbol(sym.EQ); }
"(" { return new Symbol(sym.LPAREN);}
")" { return new Symbol(sym.RPAREN); }
"[" { return new Symbol(sym.LSQBRACKET);}
"]" { return new Symbol(sym.RSQBRACKET); }
\\ { return new Symbol(sym.BSLASH); }
":=" { return new Symbol(sym.ASSIGN); }
":" { return new Symbol(sym.COLON); }
"->" { return new Symbol(sym.CONSTRUCTS); }
"," { return new Symbol(sym.COMMA); }
"'" {return new Symbol(sym.TRIV); }
```

```
"^" { return new Symbol(sym.POW); }
"@tw" { return new Symbol(sym.TIMEWORLD); }
"@w"({INTNUMBER})?"t"({INTNUMBER})? { return new Symbol(sym.INTDESC,yytext().Substring(2)); }
"Tuple" { return new Symbol(sym.TUPLE); }
"List" { return new Symbol(sym.LIST); }
(ForAll)|(Exist)|(Every)|(Some)|(True)|(False)|(Or)|(And)|(Not) { return new Symbol(sym.KEYWORD,
yytext()); }
{BASETYPE} {return new Symbol(sym.BASETYPE, yytext()); }
[1-9] { return new Symbol(sym.NONZERODIGIT, yytext()); }
{REALNUMBER} { return new Symbol(sym.NUMBER, yytext()); }
{UPLNAME} { return new Symbol(sym.UPLNAME,yytext()); }
{LOWLNAME} { return new Symbol(sym.LOWLNAME,yytext()); }
{WHITESPACE} { break; }
\. {return new Symbol(sym.TERMINATION); }
. { System.Console.WriteLine("Illegal character: "+yytext());break; }
```

Soubor obsahuje tři části, vzájemně oddělené dvěma dvojicemi znaků procenta - %%. V první části je kód, který je vložen do vygenerovaného souboru, především se zde vkládají importy hlaviček. V druhé části je místo pro speciální direktivy C#Lexu uvedeny jedním znakem procenta (zde je použitá pouze direktiva %cup), a pro definice maker pro regulární výrazy. V třetí, nejdůležitější části jsou definována pravidla pro tokeny pomocí regulárních výrazů a vytvořených maker (při použití makra musí být jeho název uveden ve složených závorkách – (UPLNAME, LOWLNAME), a k nim jsou v složených závorkách uvedeny akce v kódu, které mají být při rozpoznání daného tokenu provedeny. Většinou se jedná o vrácení tokenu, reprezentovaného třídou Symbol (název tokenu se předává v konstruktoru jako konstanta definována ve třídě sym). V případě, že potřebuje dále pracovat i s konkrétním řetězcem daného tokenu, použijeme konstruktor třídy Symbol s dvěma argumenty, kde druhým argumentem je daný řetězec, který získáme jako návratovou hodnotu funkce yytext().

Do lexikální analýzy jsem kromě klíčových slov zahrnul i pravidla pro názvy proměnných, čísla a posloupnosti bílých znaků, které byly v zadané gramatice TIL-Scriptu definovány pomocí bezkontextové gramatiky. Takovéto řetězce lze snadno vyjádřit pomocí regulárního výrazu a není třeba je rozpoznávat až během analýzy syntaktické.

3.1.2 Převod do Prologu

Než si něco řekneme o průběhu syntaktické analýzy, popíšeme si, jak budou prvky jazyka TIL-Script převedeny na predikáty jazyka Prolog pro další zpracování.

Datové typy

Názvy bázových typů odpovídají názvům v TIL-Scriptu, akorát je potřeba je uvádět v jednoduchých uvozovkách, jelikož se jedná o názvy začínající velkým písmenem. Uvozovky zajistí, že je Prolog nebude považovat za proměnné. Pro reprezentaci parciálních funkci jsem využil seznamy, kde hlava seznamu je návratový typ, a ostatní prvky jsou typy argumentů.

Převod datových typu do Prologu

1. Bázové typy – prvek báze v Prologu reprezentuje atom se stejným názvem (v uvozovkách).
2. Parciální funkci ($\alpha\ \beta_1 \ldots \beta_m$) definovanou v TIL-Scriptu převádíme na seznam $[\alpha, \beta_1, \ldots, \beta_m]$.
3. Typy $\alpha_{\tau\omega}$ jsou vždy rozepsány jako 2 vnořené seznamy – [[α, ‘Time‘],‘World‘].
4. Typ konstrukce je uveden symbolem hvězdičky, bez udání řádu, stejně jako v TIL-Scriptu.

Příklad: množina lichých unárních funkcí definovaných na reálných číslech
TIL-Script : (Bool (Bool Real)) Prolog : [‘Bool’,[‘Bool’,’Real’]]

Přiklad: propozice
TIL-Script : Bool@tw Prolog : [[‘Bool’,’Time’],’World’]

Datové typy jsou přiřazovány entitám pomocí relace *type/2*, neukládám zástupné názvy pro často používané typy, které lze v TIL-Scriptu definovat pomocí TypeDef, tady je to zbytečné, během analýzy konkrétní typ vkládám ke všem entitám. Názvy entit musí být taktéž v uvozovkách, jelikož se jedná o názvy začínající velkým písmenem. Globální proměnné a jejich typy definuji pomocí relace *globalVariable/2*. Vícenásobné definice v Prologu definuji zvlášť predikát pro každou entitu nebo proměnnou.

Převod definicí entit a proměnných z jazyka TIL-Script do Prologu (malým x proměnné, velkým X entity).

$x_1, \ldots, x_n \text{ -> } \alpha \rightarrow$ globalVariable(x_1, α).
.
.
.
globalVariable(x_n, α).

$X_1, \ldots, X_n / \alpha \rightarrow$ type(X_1, α).
.
.
.
type(X_n, α).

Příklad:
TIL-Script :
TypeDef Vlastnost:=(Bool Indiv)@tw.
Lenost,Pilnost / Vlastnost.
x,y,z/Int.

Prolog:
type(‘Lenost‘, [[[‘ Bool‘,‘Indiv‘],‘Time‘],‘World‘]).
type(‘Pilnost‘, [[[‘Bool‘,‘Indiv‘],‘Time‘],‘World‘]).
globalVariable (x,‘Int‘).
globalVariable (y,‘Int‘).
globalVariable (z,‘Int‘).

Konstrukce

Nyní je nutno říct, jakým způsobem jsou v Prologu reprezentovány konstrukce. Konstrukce ze zdrojového souboru převádím na složený term.

Proměnnou definuji pomocí atomu, reprezentujícího její název. Trivializaci reprezentuji predikátem *tr/1*. Kompozici relací *komp/2* , kde první argument je konstrukce konstruující funkci a druhý je seznam konstrukcí použitých jako argumenty dané funkce. Intenzionální sestup konstrukce *X*, v TIL-Scriptu reprezentován jako *X@wt*, rozvádím v Prologu do dvou kompozic. Uzávěr definuji pomocí relace *^/2*, kterou používám jako infixový operátor. První argument je seznam proměnných (proměnné zde reprezentujeme pomocí seznamu dvojic - název,typ) a druhý je konstrukce.

Převod konstrukcí do Prologu:

1. trivializace ’X → tr(X).
2. n-provedení ^n C → exec(n,C).
3. uzávěr [\x_1:t_1, … , x_n:t_n Y] → [[x_1,t_1], ... ,[x_n,t_n]]^Y.
4. kompozice [X X_1... X_m] → komp(X,[X_1 , ... , X_m]).
5. kompozice X@wt → komp(komp(X,[w]),[t]).

Příklad: Tom je student

TIL-Script : [\w:world [\t:time [’Student@wt ’Tom]]]
Prolog: [[w,’World’]]^[[t,’Time’]]^komp(komp(komp(tr(‘Student’),[w]),[t]),[tr(‘Tom’)]).

„1+2 = 3“
TIL-Script : [’= [’+ ’1 ’2] ’3]
Prolog : komp(tr(=),[komp(tr(+,[tr(1),tr(2)])),tr(3)]).

Konstrukce v této formě ukládám pomocí dynamického predikátu *constructionRoot/1*. Záznam konstrukce ukládám vždy jen pro „celé“ konstrukce, ne pro všechny její podkonstrukce, proto *Root*[5] v názvu.

3.1.3 Syntaktická analýza

Pro syntaktickou analýzu je krom již zmíněného nástroje C#Cup potřeba rozhraní mezi jazyky Prolog a C#, knihovna SwiPlCs.[6] Pro běh této knihovny je důležité mít nainstalovanou 64-bitovou verzi softwaru SWI-Prolog[7]. Dále je potřeba mít v systémové proměnné SWI_HOME_DIR uloženou cestu do adresáře SWI-Prologu (s názvem swipl) a do systémové proměnné PATH přidat cestu do bin adresáře ve stejné složce. Během syntaktické analýzy jsou sestavovány výše popsané převedené prvky a ukládány jako řetězce do vhodných datových struktur k pozdějšímu hromadnému uložení do báze znalostí Prologu. U objektů, kde je potřeba uložit dvojice název-datový typ jsou využity slovníky (pro proměnné, entity a typové definice), kde klíčem je název. Pro ukládání konstrukcí je využit seznam. Definice těchto slovníků a seznamů v jazyce C# vypadá následovně.

[5] Root, jelikož se v derivačním stromu konstrukce nachází nejvýše, v kořeni (viz dále).
[6] http://www.lesta.de/prolog/swiplcs/Generated/Index.aspx
[7] http://www.swi-prolog.org/

```
static Dictionary<string, string> typeDefinitions = new Dictionary<string, string>();
static Dictionary<string, string> entities = new Dictionary<string, string>();
static Dictionary<string, string> variables = new Dictionary<string, string>();
static List<string> constructions = new List<string>();
```

Údaje po syntaktické analýze jsou z těchto seznamů vkládány do báze znalostí Prologu pomocí následujících předpřipravených parametrických řetězců.

```
static string assertCon =  "assert(constructionRoot({0})).";
static string assertEntity = "assert(type('{0}',{1})).";//upperletter name, proto ''
static string assertVariable = "assert(globalVariable({0},{1})).";
```

Příkazů pro uložení je o jeden méně než kolekcí pro objekty. Je tomu tak z toho důvodu, že definice typů ve slovníku typeDefinitions jsou využity jen pro dosazování při jejich využití. Při rozpoznávání kontextů v jazyce Prolog se s nimi nepracuje. Volání těchto příkazů *assert* Prologem realizuje statická metoda třídy *PlQuery* z knihovny SwiPlCs. Jedná se o metodu *PlCall(String)*, která zavolá cíl daný vstupním řetězcem.

Ukládání konstrukcí do báze znalostí je pak velmi jednoduché:

```
foreach(string con in constructions)
            PlQuery.PlCall(String.Format(assertCon,con));
```

Ukládání entit a proměnných je již trochu složitejší, jelikož jejich typ, může být výjádřen i nepřímo přes název definice objektu uložené v slovníku typeDefinitions. Problém je také to, že v souboru může být definice uvedena až po jejím použiti(o několik řádků níže), a tedy během zpracování se konkrétní typ se v seznamu prozatím nemusí nacházet. V takovém případě namísto typu dočasně uložím název typu uvedený znakem ‘#’ pro pozdější snadné rozpoznání. Ukládání s kontrolou prvního znaku abychom rozeznali zda se jedná vypadá následujícíci mzpůsobem

```
foreach (string key in entities.Keys)
{
    if(entities[key][0]=='#')//typename beginnig with sharp
    {
        string prologType =typeDefinitions[entities[key]];
              PlQuery.PlCall(String.Format(assertEntity, key, prologType));
    }
    else//direct type definition
    {
        string prologType = entities[key];
        PlQuery.PlCall(String.Format(assertEntity, key, prologType));
    }
}
```

Analogicky vypadá ukládádní proměnných:

```
foreach (string key in variables.Keys)
{
    if(variables[key][0]=='#')//typename beginnig with sharp
    {
        string prologType = typeDefinitions[variables[key]];
            PlQuery.PlCall(String.Format(assertVariable, key, prologType));
    }
```

```
    else//direct type definition
    {
        string prologType = variables[key];
        PlQuery.PlCall(String.Format(assertVariable, key, prologType))
    }
}
```

3.2 Zpracování v jazyce Prolog

V této části popíšeme, jakým způsobem jsou navrženy a implementovány v Prologu algoritmy pro zpracování TIL-Scriptu.Více o využití Prologu pro potřeby TIL-Scriptu a TIL viz [2] . Podrobný výklad o jazyce Prolog viz např. [8].

Zpracováním se zde myslí následující úkoly:
1. Zpracování vstupu, vytvoření databáze konstrukcí
2. Typová kontrola
3. Určení kontextu
4. Výpis do souboru XML

Než si cokoliv řekneme o samotných algoritmech, nejdříve si podrobně popíšeme, jakým způsobem budeme pracovat s konstrukcemi, typy a proměnnými v jazyce Prolog. Za účelem ukládání konstrukcí je vytvořen dynamický predikát *construction/10*. Uložené konstrukce ve formě složeného termu, které jsou výstupem ze syntaktické analýzy, bude potřeba přepracovat a rozložit. Na rozdíl od předchozí formy nám relace *construction* umožní ukládat si informace jako konstruovaný typ a výskyt, bez nutnosti opakovaných výpočtů. Relaci *construction* vytvářím a ukládám nejen pro celé konstrukce ve vstupním souboru, ale také pro všechny jejich podkonstrukce. Tyto záznamy jsou přes vzájemné odkazy poskládány do logického celku, v tomto případě stromové struktury. Konstrukce, které jsou zapsány ve zdrojovém souboru, se nacházejí vždy v kořeni stromu a budu dále nazývat kořenovými konstrukcemi. Nyní přejdeme k popisu jednotlivých argumentů predikátu *construction(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10)*.

X1 – ID: každá konstrukce má svůj jedinečný identifikátor, který ji bude identifikovat. Jedná se o kód tvořený čísly a znaky „#“, který kromě jedinečné identifikace také vyjadřuje, na jaké pozici je konstrukce ve stromu umístěna. Struktura ID je podrobně popsána v následující kapitole.

X2 - Typ konstrukce: o jaký typ konstrukce se jedná - trivialisation, variable, closure, composition, execution.

X3 - ID rodiče: ID nejbližší nadřazené konstrukce. V případě že taková není, nabývá hodnoty none.

X4 - ID potomků: jedná se o seznam potomků. Potomkem jsou myšleny nejbližší podkonstrukce. V případě atomických konstrukcí je seznam prázdný.

X5 - Proměnné: v případě uzávěru je zde uložen seznam typovaných proměnných, tj. seznam uspořádaných dvojic název-typ vyjádřených dvouprvkovým seznamem. Pokud se nejedná o uzávěr, je seznam prázdný.

X6 - Počet provedení: pokud se jedná o provedení, je zde uložen počet, v případě jiných druhů konstrukce je zde uložena jednička.

X7 - Atomická konstrukce: pole využívané u atomických konstrukcí, v případě proměnné je zde uložen její název, v případě trivializace nebo provedení je zde název použitého objektu.

X8 – TIL-Script: vyjádření dané konstrukce v TIL-Scriptu

X9 - Datový typ: přesněji řečeno typ objektu konstruovaného konstrukcí

X10 – Výskyt: v jakém kontextu se konstrukce vyskytuje – intenzionální / extenzionální / hyperintenzionální

Poslední tři údaje u konstrukcí nejsou známy okamžitě po zpracování souboru, nýbrž později dopočítány. Do té doby nabývají hodnoty *unknown*.

Díky relaci *construction* je možno se jednoznačně a celkem jednoduše dotazovat na uložené konstrukce. V dotazech na prvním místě jako ID dosadíme nenavázanou proměnnou jako výstup. Pro získání všech konstrukcí vyhovujících daným podmínkám je třeba využít backtrackingu.

Například:
```
construction(ID,composition,_,_,_,_,_,_,_,_).
```
Vrací ID všech kompozic.

```
construction(ID,trivialisation,_,[_],_,_,_,_,_,_).
```
Vrací ID všech trivializací konstrukce. Je důležité určit, že chce trivializace s právě jedním potomkem, nezajímají nás trivializace objektů z analýzy, jejichž seznam potomků je prázdný.

```
construction(ID,_,none,_,_,_,_,_,_,_).
```
Vrací ID všech kořenových konstrukcí.

Relaci *construction* lze samozřejmě používat taktéž jako dotaz na atributy konkrétní konstrukce. Když známe její ID.

Například:
```
construction('#4#2',X1,X2,X3,X4,X5,X6,X7,X8,X9).
```
Dotaz uloží do proměnných X1-X9 atributy konstrukce s id #4#2.

Proměnné jsou ukládány jako dvojice jméno-typ pomocí dynamických predikátů *localVariable/2* a *globalVariable/2*. Globální proměnné jsou již zpracovány a uloženy během syntaktické analýzy. Jejich záznamy jsou během zpracování Prologem neměnné. Oproti tomu lokální proměnné jsou ukládány a mazány během průchodů konstrukcemi. Uložení proměnných, ke kterému dochází při průchodu uzávěrem, vypadá následovně.

```
%uloženi a smazáni lokálních proměnných (v uzávěru)
saveVariables([]).
saveVariables([[Name,Type]|T]):-C=..[localVariable,Name,Type],
                                 asserta(C),saveVariables(T).
```

Použiti *asserta* namísto *assert(assertz)* je zde důležité vzhledem k tomu, že v případě 2 proměnných se stejným názvem platí vždy ta definovaná naposled a Prolog prochází databázi shora dolů. Naposled definovaná proměnná se stejným názvem je v databázi nejvýše, a bude tedy první nalezena.

Při analýze konstrukce během procházení pak potřebujeme zjistit, zda existuje proměnná s daným názvem, případně objekty jakého typu konstruuje. Tuto informaci zjistíme promocí metody *variable(+Name,-Type)*, která prohledává lokální a globální proměnné. Při nalezení proměnné s daným názvem vrátí její typ.

```
variable(X,Y):-localVariable(X,Y),!.
variable(X,Y):-globalVariable(X,Y),!.
```

Řez nám zde zajistí to, že první nalezená proměnná s odpovídajícím názvem bude jediným výsledkem dotazu.

Ještě si pro úplnost ukážeme metodu *deleteVariables(+ListOfVariables)*.

```
deleteVariables([]).
deleteVariables([[Name,Type]|T]):-C=..[localVariable,Name,Type],
                                   retract(C),!, deleteVariables (T).
```

Metoda je velmi podobná té pro uložení proměnných, jediným rozdílem je použití řezu. *Retract* totiž oproti příkazu *assert* nemá jen jedno plnění, ale při backtrackingu maže další odpovídající záznamy. Tento nežádoucí efekt je řezem odstraněn.

Typy objektů jsou podobně jako globální proměnné zpracovány během syntaktické analýzy a během zpracování Prologem se nemění. Jsou využívány k dotazu, zda daný objekt existuje a případně jakého je typu. Na objekt *X* se ptáme následujícím způsobem.

```
type(X,Type).
```

Než začneme popisovat konkrétní algoritmy, nejprve stručně popíšeme algoritmus procházení konstrukcí způsoben do hloubky, který je pro několik algoritmů společný.

Název: depthFirstSearch
Vstup: konstrukce C

Zpracuj konstrukci C
P=potomci konstrukce C
Pro každou konstrukci D v P proveď
depthFirstSearch(D)

V některých algoritmech, kde zpracování konstrukce závisí na výsledku zpracování potomků, probíhá zpracování konstrukce až po skončení volání potomků. Dále některé algoritmy, které využívají prohledávání do hloubky, mohou každý typ konstrukce zpracovávat jiným způsobem (například při typové kontrole). V tom případě zpracování konstrukce mohou předcházet podmínky pro zjištění, který typ konstrukce je právě zpracováván.

Jako přiklad uvedu analýzu výrazu: „Tilman hledá poslední desetinné místo čísla π“ v TIL-Scriptu.

```
Tilman/Indiv.
Seek / ( Bool Indiv *) @tw.
Lastdec/(Int Real).
Pi/Real.
[\w: World [\t:Time['Seek@wt 'Tilman '['Lastdec 'Pi]]]].
```

Derivační strom znázorňující strukturu konstrukce ukazuje Obr. 1.

Jednotlivé uzly jsou očíslovány podle pořadí, v jakém budeme uzly procházet během prohledávání do hloubky.

Jednoduchá implementace algoritmu procházení do hloubky v Prologu může vypadat následovně (jako zpracování konstrukce provedeme výpis jejího ID).

```
depthFirstSearch(ID):-writeln(ID),      construction(ID,_,_,DescIDs,_,_,_,_,_,_),
       call2(DescIDs,depthFirstSearch).
```

Rekurzivní volání je spouštěno v cyklu metody *depthFirstSearch/0* pro každou kořenovou konstrukci pomocí metody konstruktivního selhání následujícím způsobem:

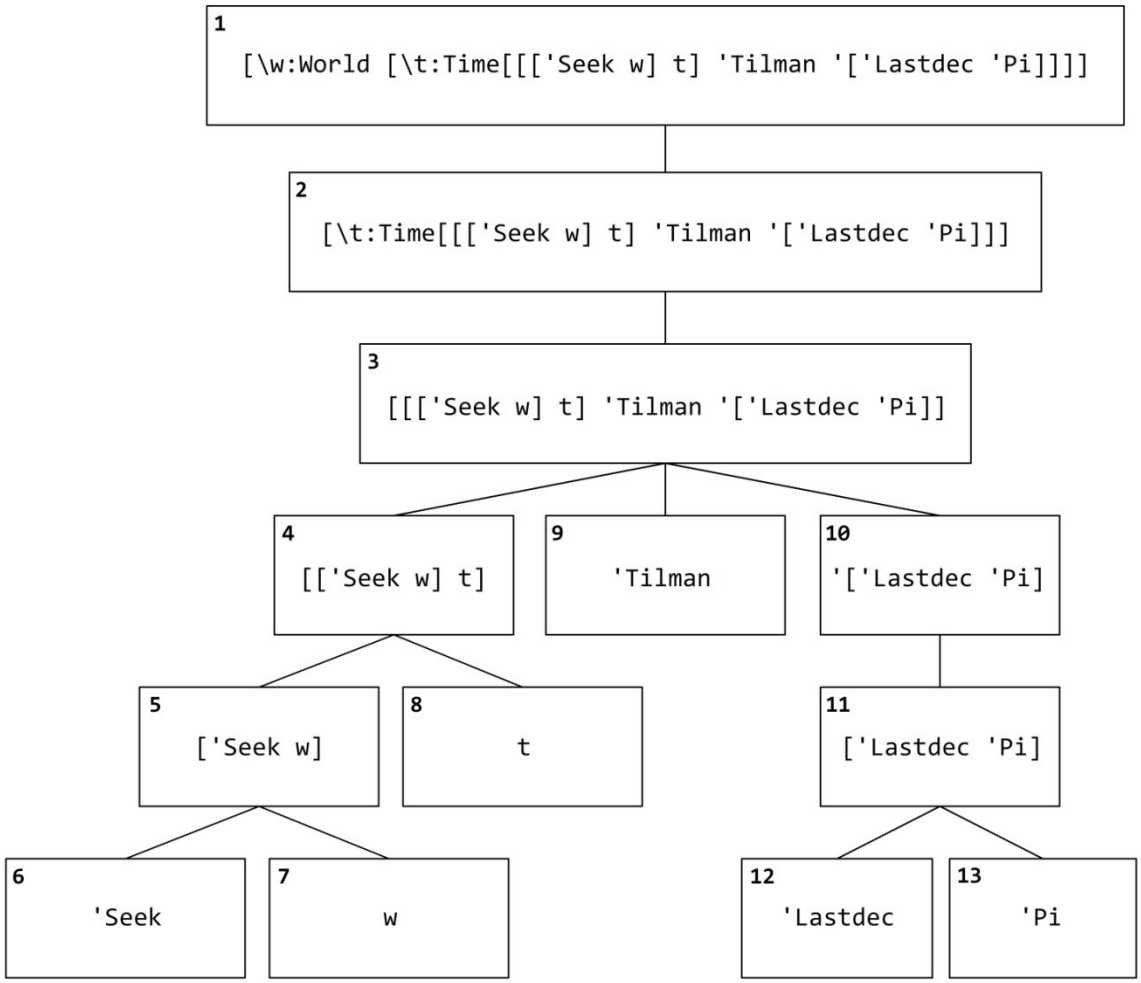
```
depthFirstSearch:-construction(ID,_,none,_,_,_,_,_,_,_),depthFirstSearch(ID),fail.
depthFirstSearch.
```

Tam, kde potřebujeme zavolat algoritmus průchodu do hloubky pro všechny potomky konstrukce, používáme predikát *call2(+SeznamPotomků,+Predikát)*, který postupně zavolá predikát předaný jako argument na každý prvek seznamu. V případě atomických konstrukcí je první seznam argumentů prázdný a predikát *call2* automaticky skončí úspěchem. Důležitým předpokladem je, že seznam neobsahuje instanciované proměnné. Kvůli selhání na konci každého volání je také nutné, aby predikát byl ošetřen před nežádoucím backtrackingem, tj. aby při backtrackingu nacházel jen platná řešení.

```
call2(L,P):-member(M,L),Call=..[P,M],Call,fail.
call2(_,_).
```

V našem případě konkrétní volání *call2(['#4#1','#4#2','#4#3'],depthFirstSearch)* zavolá následující cíle:

```
depthFirstSearch ('#4#1')
depthFirstSearch ('#4#2')
depthFirstSearch ('#4#3')
```



Obr. 1 – ukázka derivačního stromu

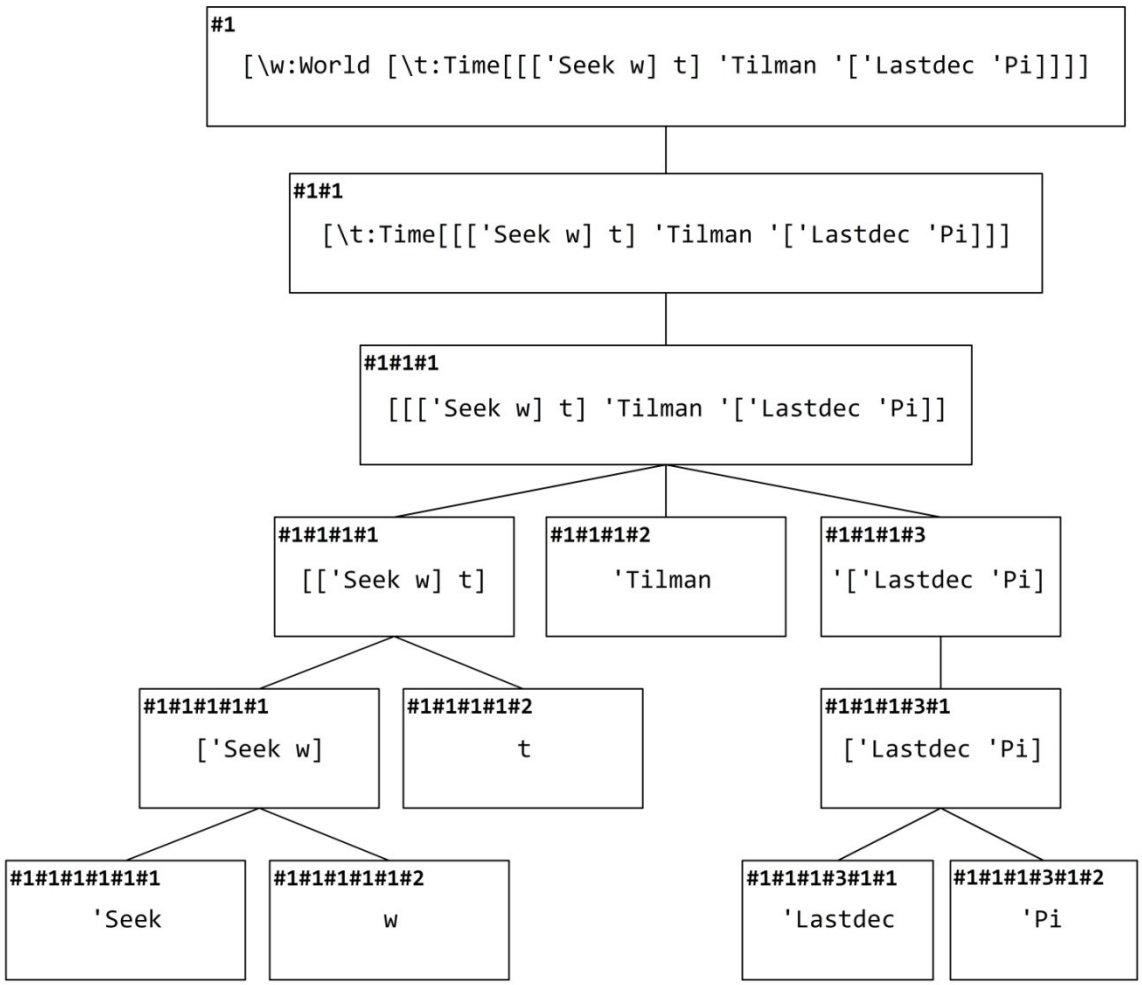
3.2.1 Vytvoření databáze konstrukcí

Nejdříve je potřeba přepracovat konstrukce ve formě složeného termu uložené pomocí predikátu *contructionRoot*. Je potřeba také určit pro každý záznam jedinečný identifikátor – ID, který také nese informaci o poloze uzlu. Nyní si popíšeme formát ID.

ID je zřetězením dvojic *#N*, kde *N* je přirozené číslo. Počet těchto dvojic určuje hloubku daného uzlu. Číslo *N* určuje, o kolikátého potomka nadřazené konstrukce se jedná (případně pořadí ve zdrojovém souboru u kořenových konstrukcí). *N* kořenových konstrukcí (tj. uzly s hloubkou jedna) má po řadě ID *#1, #2 ... #N*. *M* potomků konstrukce s ID *#N* mají ID #N#1, #N#2 … #N#M atd.
Jako příklad určení ID je na Obr. 2 uveden předchozí strom s jednotlivými ID.

ID je dynamicky ukládáno a mazáno během průchodu konstrukcemi. ID jsou ukládány pomocí dynamického predikátu *id/1*. Pro práci s ID máme metody *getID(-NovéID)* a *getID(-NovéID,+Pořadí)*, které nám krom vytvoření nového ID pro aktuálně zpracovávaného potomka nové ID uloží. Po zpracování potomka a jeho podkonstrukcí je ID smazáno. Záznamy *id/1* se nepřepisují, nýbrž ukládají na vrch databáze pomocí *asserta*, a tedy aktuální ID je pouze první záznam.

Algoritmus postupně prochází do hloubky složený term konstrukce, rozkládá ho, a ukládá jednotlivé podkonstrukce. V každém kroku algoritmu zpracováváme konstrukci C, pro kterou je již uložen záznam (o uložení se stará rodičovská konstrukce), a získáme její potomky X_1-X_N. (zatím nezpracované ve formě termů). Pokud je konstrukce C uzávěr, uložíme λ-proměnné daného uzávěru. Postupně pro každého potomka X_i získáme nové ID, rozpoznáním a rozložením termu potomka X_i získáme potřebné atributy konstrukce, a uložíme je již jako záznam relace *construction/10*. Poté rekurzivním zavoláním pokračujeme ve zpracovávání potomka X_{i+1} a jeho případných podkonstrukcí. Po vyhodnocení podkonstrukcí smažeme λ-proměnné. Poté konstrukce aktualizuje svůj seznam potomků, namísto složených termů již uloží jejich ID.



Obr. 2 - derivační strom konstrukce se zpracovanými ID

Implementace v jazyce Prolog se skládá z predikátů *processID(+ID)* a *processID/0*.

```
%findall využito k zavolání složeného cíle pro všechny potomky (jejich ID sbírá do
seznamu IDs)
%pro získání všech potomků ze seznamu je využito nth1 namísto member(potřebujeme
pořadí prvku)
%uložení potomka zajistí saveID(+TermKonstrukce,+NovéID,+IDpředka)
%aktualizaci potomků zpracovavane konstrukce zajistí
updateCon(+ID,+SeznamIDPotomků).

processID(X):-construction(X,_,_,Descs,Variables,_,_,_,_,_),
              saveVariables(Variables),
              findall(ID,
                        (nth1(Index,Descs,Con),getID2(ID,Index),
                        saveID(Con,ID,X),processID(ID)),
                        IDs),
              deleteVariables(Variables),
              updateCon(X,IDs).

%volání na kořenové konstrukce
processID:-findall(Con,constructionRoot(Con),ConstructionRoots),
              nth1(Index,ConstructionRoots,Member),
              getID2(NewID,Index),saveID(Member,NewID,none),
              processID(NewID),
              fail.
```

3.2.2 Typová kontrola

Důležitým údajem, který potřebujeme zjistit, je typ konstruovaného objektu všech konstrukcí. Algoritmus prochází konstrukce do hloubky. Oproti předchozím algoritmům algoritmus pro typovou kontrolu a uložení typů využívá výstupní proměnnou, která vrací typ objektu konstruovaného danou konstrukcí (pro zpracování rodičem). V každém kroku algoritmu pro zpracovávanou konstrukci *C* buď určíme výsledný typ přímo (pokud se jedná o konstrukci bez potomků), nebo nejprve zpracujeme a vyhodnotíme potomky a z typů objektů, které konstruují, sestavíme typ konstruovaný konstrukcí *C*. Pokud nastane případ, že je konstrukce *C* nevlastní z důvodu špatného typování nebo z důvodu neexistence proměnné či trivializovaného objektu, namísto datového typu ponechá původní hodnotu *unknown*. Parcialita je propagována nahoru, nevlastní jsou také konstrukce, které obsahují *C* mezi konstituenty, a tedy u všech zůstává uložena hodnota *unknown*.

Samotná typová kontrola se týká jen Kompozice a probíhá podle definice konstrukcí. Je potřeba zjistit, zda typy konstruované konstrukcemi $Y_1 \dots Y_n$ Kompozice $[X\ Y_1 \dots Y_n]$ odpovídají typům $\beta_1 \dots \beta_m$ funkce $(\alpha\beta_1 \dots \beta_m)$ konstruované konstrukcí *X*. Vzhledem k přehlednosti a lepší možnosti výpisu chyb typová kontrola probíhá jako samostatná metoda.

Procházení konstrukcemi, skládání a ukládání typu konstruovaného objektu je realizováno metodou *determineType(+ID,-DatovýTyp)*, která prochází konstrukce, a ukládá výsledné typy. Typy jsou ukládány pomocí metody *saveDataType(+ID,+DatovýTyp)*. Další důležitou novou metodou je metoda realizující typovou kontrolu *typeControl(+SeznamID,+SeznamTypů)*.

determineType/2 vypadá následovně:

```
%proměnné
determineType(ID,DataType):-construction(ID,variable,_,[],_,_,C,_,_,_),
                                variable(C,DataType),
                                saveDataType(ID,DataType),!.

%trivializace entit z analýzy
determineType(ID,DataType):-construction(ID,trivialisation,_,[],_,_,X,_,_,_),
                                type(X,DataType),
                                saveDataType(ID,DataType),!.
%trivializace konstrukce
determineType(ID,DescID):-construction(ID,trivialisation,_,[DescID],_,_,_,_,_,_),
                                determineType(DescID,_),
                                saveDataType(ID,DescID),!.

determineType(ID,DataType):-
construction(ID,composition,_,[FunID|ArgIDs],_,_,_,_,_,_),
                                determineType(FunID,[DataType|LstOfTypes]),
                  kontrola(ArgIDs,LstOfTypes),
                  saveDataType(ID,DataType),!.

%totypelist převádí seznam typovaných proměnných na seznam jejich typů
determineType(ID,[AlfaObject|LstOfTypes]):-
construction(ID,closure,_,[DescID],Variables,_,_,_,_,_),
                  saveVariables(Variables),
                  determineType(DescID,AlfaObject),
                  deleteVariables(Variables),
                  totypelist(Variables,LstOfTypes),
                  saveDataType(ID,[AlfaObject|LstOfTypes]),!.


%úklid proměnných po selhání(pokud se jednalo o uzávěr)
determineType(ID,_):-construction(ID,closure,_,_,Variables,_,_,_,_,_),
                                deleteVariables(Variables),fail.

%dvoji provedeni trivializace konstrukce
determineType(ID,Y):-construction(ID,execution,_,[DescID],_,2,_,_,_,_),
                        determineType(DescID,Type),
              construction(Type,_,_,_,_,_,_,_,_,Y,_),
                        saveDataType(ID,Y),!.

determineType(X,_):-write('nerozpoznano: '),writeln(X),fail.
```

determineType/2 je spouštěno *determineType/0* pro každou kořenovou konstrukci.:

```
determineType:-construction(ID,_,none,_,_,_,_,_,_,_),determineType(ID,_),fail.
```

Metoda pro typovou kontrolu *typeControl(+SeznamID,+SeznamTypů)* v Prologu vypadá následovně.

```
typeControl([],[]).
typeControl([_|_],[]):-writeln('chyba - moc parametru'),fail.
typeControl([],[_|_]):-writeln('chyba - malo parametru'),fail.
```

```
%pokud očekáváme konstrukci
typeControl([H1|T1],['*'|T2]):- determineType(H1,T),
                                isConstruction(T),!,
                                typeControl(T1,T2).

typeControl([H1|T1],[H2|T2]):-determineType (H1,T),
                                T=H2,!,
                                typeControl(T1,T2).

%chybový výpis, selhání
kontrola([H1|_],[H2|_]):-write(H1),write(' nekonstruuje '),write(H2),
                         writeln(' - chyba'),fail.
```

3.2.3 Rozpoznávání kontextů

Když chceme rozhodnout o kontextu, v jakém se vyskytuje daná konstrukce, je třeba nejprve rozhodnout, zda je konstrukce zmíněna nebo užita jako konstituent. Pokud je v konstrukci *C* podkonstrukce *D* zmíněna, je samotná konstrukce *D* objektem, o kterém se vypovídá, a není ji potřeba provést při provádění konstrukce *C*. V takovém případě má konstrukce *D* a všechny její podkonstrukce hyperintenzionální výskyt. Pokud je konstrukce *D* v *C* užita jako konstituent, bude třeba dále rozhodnout, zda se jedná o extenzionální či intenzionální kontext. Zmínění konstrukce docílíme její trivializací, tím de facto zabráníme jejímu provedení. Nyní uvedu formální definici zmiňování a užívání konstrukcí. Definice v této kapitole jsou citovány z [1].

Definice 6 (*konstrukce zmíněna vs. užita jako konstituent*).
Nechť *C* je konstrukce a *D* podkonstrukce *C*.

i) Je-li *D* identická s *C* (tj. ${}^0C = {}^0D$), pak *výskyt D* je *užit v C jako konstituent.*
ii) Je-li *C* identická s $[X_1\ X_2 \ldots X_m]$ a *D* je jedna z konstrukcí $X_1, X_2, \ldots, X_m$, pak *výskyt D* je *užit v C jako konstituent*.
iii) Je-li *C* identická s $[\lambda x_1 \ldots x_m\ X]$ a *D* je *X*, pak *výskyt D* je *užit v C jako konstituent.*
iv) Je-li *C* identická s ${}^1X$ a *D* je *X*, pak *výskyt D* je *užit* v *C jako konstituent*.
v) Je-li *C* identická s ${}^2X$ a *D* je *X*, nebo ${}^0D$ se vyskytuje jako konstituent *X* a tento výskyt *D* je konstituentem konstrukce *Y v*-konstruované *X*, pak *výskyt D* je *užit v C jako konstituent*.
vi) Je-li *výskyt D užit jako konstituent* výskytu konstrukce *C'* a tento výskyt *C'* je *užit v C jako konstituent*, pak *výskyt D* je *užit v C jako konstituent.*
vii) Jestliže výskyt podkonstrukce *D* konstrukce *C* není užit v *C* jako konstituent, pak je výskyt *D zmíněn v C*.
viii) Výskyt podkonstrukce *D* konstrukce *C* je v *C* užit nebo zmíněn *C* pouze dle i)-vii).

Bod v) uvádí případ, kdy dvojí provedení ruší účinek Trivializace. Např. v konstrukci ${}^{20}X$ je konstrukce *X* užita, protože dvojí provedení účinek Trivializace zrušilo. Dvojí Provedení však ruší účinek jen jedné, nejbližší Trivializace, v případě ${}^{200}X$ je konstrukce *X* zmíněna.

Algoritmus rozpoznávající, zda je konstrukce zmíněna nebo užita, prochází postupně všechny konstrukce do hloubky. Velmi důležitý je argument S, tj. proměnná, která může nabývat dvou hodnot: „used“ a „mentioned“. Na začátku algoritmu tento vstupní argument S nabývá hodnoty „used”, jelikož každá kořenová konstrukce je užita.

Název: determineHyperintensional
Vstup: konstrukce C, konstanta S určující, zda se nacházíme ve zmíněné nebo užité konstrukci

Pokud S=”mentioned”
    Ulož hyperintenzionální kontext konstrukce C
Pokud S=”used” a současně je C trivializace konstrukce tvaru ^{0}D
    Zavolej determineHyperintensional(D,”mentioned”)
Pokud S=”mentioned” nebo C není trivializace konstrukce tvaru ^{0}D
    P=potomci konstrukce C
    Pro každou konstrukci D v P proveď
        determineHyperintensional (D,S)

Metoda *determineHyperinstesional(+ID,+UsedOrMentioned)* v Prologu vypadá následovně:

```
%zmíněno – uložím výskyt, selžu a jdu dál
determineHyperintensional(X,mentioned):-
                    saveOccurrence(X,'Hyperintensional'),fail.

%trivializace, kde je jednoprvkový seznam potomku = trivializace konstrukce,
%konstrukce v podstromu jsou zmíněny – mentioned
determineHyperintensional(X,used):-
                    construction(X,trivialisation,_,[DescID],_,_,_,_,_,_),
                    determineHyperintensional(DescID,mentioned),!.

% nezávisle na tom zda je zmíněno nebo užito, zavolám rekurzi pro podstrom
determineHyperintensional(X,Y):-
                    construction(X,_,_,DescIDs,_,_,_,_,_,_),
                    call2(DescIDs,determineHyperintensional,Y).
```

Řez v druhém pravidle zamezí zbytečnému druhému průchodu podstromu zmíněné konstrukce. Je jistě zbytečné po úspěšném průchodu a uložení konstrukcí v hyperintenzionálním kontextu stejné konstrukce procházet znova jakoby se jednalo o konstrukce užité (přechod na třetí pravidlo).

V třetím pravidle je rekurzivní volání je realizováno pomocí *call2/3* namísto dřivějšího *call2/2*. Potřebujeme totiž předat další argument, který nám určí jesli voláme na zmíněnou nebo užitou podkontrukci. Tento argument bude stejný u všech volání v cyklu.

```
call2(L,P,A):-member(M,L),Call=..[P,M,A],Call,fail.    %zavolej a pak rekurze
call2(_,_,_).
```

V našem případě volání *call2([‘#4#1‘,‘#4#2‘,‘#4#3‘],determineHyperintensional,used)* zavolá následující cíle:

```
determineHyperintensional('#4#1',used)
determineHyperintensional('#4#2',used)
determineHyperintensional('#4#3',used)
```

Volání pro všechny kořenové konstrukce pomocí *determineHyperinstesional/0*:

```
determineHyperintensional:-construction(ID,_,none,_,_,_,_,_,_,_),
                                determineHyperintensional(ID,used),fail.
determineHyperintensional.
```

V našem příkladu na obrázku 1 jsou v hyperintenzionálním kontextu konstrukce 11, 12 a 13.

Teď, když víme o tom, které konstrukce jsou v hyperintenzionálním kontextu, potřebujeme rozhodnout o kontextu konstrukcí užitých. Ty se mohou vyskytovat extenzionálně, nebo intenzionálně. Určení kontextu probíhá ve 2 krocích, první je třeba zjistit supozici tj. jakým způsobem je konstrukce užita a poté zkontrolovat možné přebití nižšího kontextu vyšším pomocí procedury λ–abstrakce. Nyní uvedu formální definici supozice a popíšu návrh implementace.

Definice 7 (*intenzionální* / *extenzionální supozice*).

i) Nechť C je proměnná nebo Trivializace objektu typu řádu 1, a nechť D je identická s C, $D \rightarrow_v (\beta_1 \ldots \beta_n)$, $n \geq 1$. Pak D *se vyskytuje v* C *s* $(\beta_1 \ldots \beta_n)$-*intenzionální supozicí*.

ii) Nechť C je Uzávěr $[\lambda x_1 \ldots x_m\, X]$, $x_1 \rightarrow_v \beta_1, \ldots, x_m \rightarrow_v \beta_m$, $X \rightarrow_v \alpha$. Pak:
   1. Je-li D identická s C, pak D *se vyskytuje v* C *s* $(\alpha\beta_1 \ldots \beta_m)$-*intenzionální supozicí*.
   2. Je-li D konstituentem X, pak D *se vyskytuje v* C *se* stejnou *supozicí* jako D v X.

iii) Nechť C je Kompozice $[X\, Y_1 \ldots Y_m]$, $m \geq 1$, a $X \rightarrow_v (\alpha\beta_1 \ldots \beta_m)$, $Y_1 \rightarrow_v \beta_1, \ldots, Y_m \rightarrow_v \beta_m$. Pak:
   1. Je-li D identická s C, pak D *se vyskytuje v* C *s* α-*intenzionální supozicí.*
   2. Je-li D identická s X, pak D *se vyskytuje v* C *s* $(\beta_1, \ldots, \beta_m)$-*extenzionální supozicí*.
   3. Je-li D konstituentem X, který není identický s X nebo je-li D konstituentem Y_i $(1 \leq i \leq m)$, pak D *se vyskytuje v* C *se* stejnou *supozicí* jako D v X nebo Y_i.

iv) Nechť C je ${}^1X$ nebo ${}^2X$, kde X je konstrukce. Pak konstituenty X *se vyskytují v* C se stejnou *supozicí* jako v X.

v) Nechť C je ${}^1X$, kde X je objekt typu řádu 1, a nechť D je C. Pak D *se vyskytuje v* C s *extenzionální supozicí.*

vi) Nechť C je ${}^2X$, kde X je objekt typu řádu 1 nebo X *v*-konstruuje objekt typu řádu 1, a nechť D je C. Pak D *se vyskytuje v* C s *extenzionální supozicí.*

vii) Výskyt konstrukce s *intenzionální / extenzionální supozicí v C je pouze dle* (i) – (v).

Tato definice je složitá, pro algoritmus je mnohem lepší vyjít z důsledků této definice, které jsou rovněž uvedeny v [1]:

Důsledek. Konstituent D se vyskytuje v C s extenzionální supozicí pouze v těchto případech:

i) C je Kompozice $[D\, Y_1 \ldots Y_m]$

ii) C je (Dvojí) Provedení ${}^1[D\, Y_1 \ldots Y_m]$, ${}^2[D\, Y_1 \ldots Y_m]$

iii) D je identická s C a C je ${}^1X$, kde X je objekt typu řádu 1

iv) D je identická s C a C je ${}^2X$, kde X je objekt typu řádu 1 nebo X *v*-konstruuje objekt typu řádu 1.

Tyto důsledky však mluví pouze o supozici v nejbližší nadřazené kompozici (nebo ${}^{1}X$, ${}^{2}X$ vůči sobě samým, avšak díky rekurzivnímu kroku v definici 2 bodech ii) 2, iii) 3. a iv) pokaždé dojdeme k případu, kdy určujeme supozici vůči nebližší nadřazené konstrukci nebo sobě samé.

Schází nám však postup, jak zjistit, zda je daný objekt typu řádu jedna či nikoliv. Algoritmus, který nám tuto informaci zjišťuje, vychází z definice **1**. Typ řádu jedna je buď prvkem báze, nebo parciální funkcí typů řádů jedna.

Název: typeOfOrder1
Vstup: objekt X
Otázka: je X objekt typu řádu jedna ?
Pokud je X bázový typ
    výsledek je ano
Pokud je X parciální funkce ($\alpha_1 \ldots \alpha_n$)
        Pokud je n >1
         Pokud jsou ($\alpha_1 \ldots \alpha_n$) typy řádu 1
              Výsledek je ano
Výsledek je ne

Implementace v jazyce Prolog vypadá následovně

```
base('Indiv').
base('Bool').
base('Int').
base('Real').
base('Time').
base('World').
base('String').

%typradu1(+co)
%-------------------------------------------------------%
typeOfOrder1(X):-base(X),!.       %bud bázový typ nebo fce
typeOfOrder1(X):-list(X),length(X,L),L>1,partialMappingOfOrder1(X).
%-------------------------------------------------------%

%parcialnifceradu1(+co)
%-------------------------------------------------------%
partialMappingOfOrder1([]).
partialMappingOfOrder1([H|T]):-typeOfOrder1(H),partialMappingOfOrder1(T).
%-------------------------------------------------------%
```

Algoritmus pro získání konstrukcí s extenzionální supozicí vypadá následovně:
Název: extSupositionConstructions
Výstup: množina konstrukcí s extenzionální supozicí

Pro každou kompozici C ve tvaru [$X\ Y_1 \ldots Y_m$]; proveď
Pokud se C nevyskytuje v hyperintenzionálním kontextu
Přidej konstrukci X do výsledku
Pro každé provedení E ve tvaru ^{1}X nebo ^{2}X, kde X je objekt řádu jedna proveď
Pokud se E nevyskytuje v hyperintenzionálním kontextu
Přidej konstrukci E do výsledku
Pro každé provedení E ve tvaru ^{2}X, kde X v-konstruuje objekt řádu jedna proveď
Pokud se E nevyskytuje v hyperintenzionálním kontextu
Přidej konstrukci E do výsledku

Pro zmíněný algoritmus můžeme sestavit v Prologu metodu *extSupositionConstructions(-ID)*, která vypadá následovně:

```
extSupositionConstructions(ID):-
construction(_,composition,_,[ID|_],_,_,_,_,_,unknown).

extSupositionConstructions(ID):-
construction(ID,execution,_,[],_,_,exec(_,X),_,_,unknown),
                                type(X,T),typeOfOrder1(T).

extSupositionConstructions(ID):-
construction(ID,execution,_,[DescID],_,_,_,_,_,unknown),
    construction(DescID,_,_,_,_,_,_,_,_,DataType,_), typeOfOrder1(DataType).
```

Je důležité také určit, že chceme ty konstrukce, které mají v poli *výskyt* uloženou hodnotu *unknown*. Nezajímají nás zpracované konstrukce, které již mají uložen hyperintenzionální výskyt.

V našem případě se v extenzionální supozici vyskytují uzly 4,5 a 6. Ostatní uzly, které nebyly zmíněny a nemají extenzionální supozici, mají supozici intenzionální. Supozici však nebudu ukládat. Jak již bylo řečeno, určení kontextu probíhá ve 2 fázích, a určení supozice je pouhým mezivýsledkem, se kterým budeme dále pracovat.

Náš dosavadní výsledek vypadá takto:
Hyperintenzionální výskyt: konstrukce 11,12,13
Extenzionální supozice: konstrukce 4, 5 a 6.
Intenzionální supozice: konstrukce 1,2,3,7,8 a 10

Nyní je potřeba zkontrolovat genericitu a vyvodit závěr.

Než si uvedeme definici genericity, je třeba si definovat co je atomická konstrukce.

Definice 8 (atomická konstrukce) *Konstrukce C* je *atomická,* jestliže *C* neobsahuje žádný jiný konstituent než sama sebe.

Důsledek. Konstrukce *C* je atomická, jestliže *C* je

i) proměnná; nebo
ii) Trivializace ${}^0X$, kde *X* je entita libovolného typu, i konstrukce; nebo
iii) Provedení ${}^1X$ nebo Dvojí provedení ${}^2X$, kde *X* je objekt typu řádu 1, tj., ne-konstrukce.

Definice 9 (*generický* / *negenerický kontext*)

i) Nechť *C* je atomická konstrukce. Pak *C se vyskytuje v negenerickém kontextu C*.
ii) Nechť *C* je Uzávěr $[\lambda x_1 \ldots x_m X]$; $x_1 \rightarrow_v \gamma_1, \ldots, x_m \rightarrow_v \gamma_m$.
    a) Jestliže *D* je *C* a *X* se vyskytuje v negenerickém kontextu *X*, pak *D se vyskytuje v* $(\gamma_1,\ldots,\gamma_m)$-*generickém intenzionálním kontextu C.*
    b) Jestliže *D* je *C* a *X* se vyskytuje v (β)-generickém intenzionálním kontextu *X* pro nějaký typ β, pak *D se vyskytuje v* $((\gamma_1,\ldots,\gamma_m)\beta)$-*generickém intenzionálním kontextu C.*
    c) Jestliže *D* je konstituent *X* a *D* se vyskytuje v negenerickém kontextu *X*, pak *D se vyskytuje v* $(\gamma_1,\ldots,\gamma_m)$-*generickém intenzionálním kontextu C.*
    d) Jestliže *D* je konstituent *X* a *D* se vyskytuje v β-generickém intenzionálním kontextu *X* pro nějaký typ β, pak *D se vyskytuje v* $((\gamma_1,\ldots,\gamma_m)\beta)$-*generickém intenzionálním kontextu C.*
iii) Nechť *C* je Kompozice $[X\ Y_1 \ldots Y_m]$; $Y_1 \rightarrow_v \gamma_1, \ldots, Y_m \rightarrow_v \gamma_m$.
    a) Jestliže *X* se vyskytuje v generickém intenzionálním kontextu *X*, pak
        - Jestliže *D* je konstituent *X* a *D* se vyskytuje v $(\gamma_1,\ldots,\gamma_m)$-generickém kontextu *X*, pak *D se vyskytuje v negenerickém (extenzionálním) kontextu C*, a
        - Jestliže *D* je konstituent *X* a *D* se vyskytuje v $((\gamma_1,\ldots,\gamma_m)\beta)$-generickém kontextu *X* pro nějaké β, pak *D se vyskytuje v* (β)-*generickém (intenzionálním) kontextu C*.
    b) Jestliže *X* se vyskytuje v negenerickém extenzionálním kontextu *X*, pak *X* se vyskytuje v negenerickém extenzionálním kontextu *C* a konstituenty *X se vyskytují v C* ve stejném kontextu jako v *X*.
    c) Jestliže *D* je *C*, pak kontext, ve kterém se *D vyskytuje v C* je stejný jako kontext, ve kterém se vyskytuje *X* v *C*.
    d) Kontexty, ve kterých se mohou *vyskytovat* konstituenty Y_i $(1 \leq i \leq m)$ *v C* jsou stejné jako kontexty, ve kterých se vyskytují v Y_i.
iv) Nechť *C* je ${}^1X$ nebo ${}^2X$. Pak konstituenty *X se vyskytují v C* ve stejném kontextu jako v *X*.
v) Možné *výskyty v generickém nebo negenerickém kontextu v C* jsou pouze dle (i) – (iv).

Definice 10 (*intenzionální vs. extenzionální výskyt*). Jestliže *D* se vyskytuje s intenzionální supozicí nebo v generickém kontextu *C*, pak *D se vyskytuje v C intenzionálně*. Jinak, tj. jestliže *D* se vyskytuje s extenzionální supozicí a v negenerickém kontextu *C*, pak *D se vyskytuje extenzionálně v C*.

Z definice **10** vyplývá, že už z intenzionální supozice dané konstrukce lze usoudit, že celkový výskyt bude intenzionální, oproti tomu u konstrukcí s extenzionální supozicí je potřeba ještě ověřit genericitu, zda nedochází k přebití kontextu. Tedy nám zbývá určit, zda se v případě konstrukcí označených čísly 4, 5 a 6 skutečně jedná o extenzionální výskyt. Bude potřeba ověřit, zda se konstrukce s extenzionální supozicí nacházejí v negenerickém kontextu.

Algoritmus určení genericity probíhá téměř přesně pole definice. Pro výpočet genericity používám v Prologu metodu *genericity(+D,+C,-Genericita)*. Generický typ je reprezentován uspořádanou n-ticí, kde prvky jsou uspořádané n-tice datových typů, v případě implementace v Prologu se jedná o seznam seznamů. Velikou výhodou je to, že případ negenerického kontextu může být reprezentován prázdným seznamem. Na rozdíl od definice uvažujeme negenerický kontext spíše jako () - generický. Pak můžeme například u bodu 2)b definice 3 uvažovat, že se pod symbolem β může skrývat informace o negenerickém kontextu, tedy prázdný seznam. A když výsledný generický typ sestavíme podle definice, tj. $((\gamma_1,\ldots, \gamma_m)\beta)$ bude díky tomu de facto pouze $(\gamma_1,\ldots, \gamma_m)$. V Prologu, když se X vyskytuje v negenerickém kontextu X, pak β= [] a proměnné uzávěru $x_1 \ldots x_m$ po řadě konstruují objekty typu $\gamma_1,\ldots, \gamma_m$, a tyto typy máme uloženy v seznamu S. Výsledný generický typ lze vytvořit pomocí operátoru „|“ jako [S|β]. Tímto způsobem lze v algoritmu body ii)a a ii)b, a body ii)c a ii)d spojit v jeden případ. Podobným principem akorát s následným rozdělením seznamu reprezentujícího generický typ lze při zpracování kompozice spojit dva případy v bodu iii)a.

Dále je výhodné přehodit pořadí pravidel oproti bodům definice 3 v pravidlech pro kompozici, kde pravidlo pro bod iii)c je dobré mít na prvním místě (jelikož tento případ je velmi jednoduché zjistit a je lepší kontrolu provést hned, než až po neúspěšném hledání *D* mezi konstituenty *X*). Poté zbývající pravidla v pořadí podle definice.

Algoritmus pro výpočet generického typu vypadá následovně:

Název: genericity
Vstup: konstrukce D a C, kde D je konstituentem C
Výstup: generický typ

1. Pokud je C atomická konstrukce (a tedy C=D)
   výsledek=negenerický
2. Pokud C je uzávěr tvaru $[\lambda x_1 \ldots x_m\ X]$; $x_1 \rightarrow_v \gamma_1, \ldots, x_m \rightarrow_v \gamma_m$, pak:
   a) Pokud D = C,
      β = genericity(X,X)
      výsledek=$((\gamma_1,\ldots, \gamma_m)\beta)$

   b) Pokud D je konstituent X,
      β = genericity(D,X)
      výsledek = $((\gamma_1,\ldots, \gamma_m)\beta)$

3. Pokud C je kompozice tvaru $[X\ Y_1 \ldots Y_m]$; $Y_1 \rightarrow_v \gamma_1,\ldots, Y_m \rightarrow_v \gamma_m$.
   a) Pokud C=C
      výsledek=genericity(X,C)
   b) Pokud D je konstituent X
      G=genericity(X,X)
      Pokud G je negenericky typ
         výsledek=genericity(D,X)
      Pokud G je genericky typ $((\gamma_1,\ldots, \gamma_m)\beta)$
         výsledek = β
   c) Pokud je D konstituent konstrukce Y z Yi

výsledek = genericity(D,Y)

4. Pokud je C provedení tvaru ^{2}X nebo ^{1}X kde X je konstrukce
   Pokud je D konstituentem X
   výsledek = genericity(D,X)

Implementace v Prologu:

```
%atomická – seznam potomků je jednoprvkový, nebo trivializace konstrukce
genericity(C,C,[]):-construction(C,_,_,[],_,_,_,_,_,_).
genericity(C,C,[]):-construction(C,trivialisation,_,[_],_,_,_,_,_,_).

genericity(C,C,[TypeList|G2]):-construction(C,closure,_,[DescID],Variables,_,_,_,_,_),
                               genericity(DescID,DescID,G2),
                               totypelist(Variables,TypeList).

genericity(D,C,[TypeList|G2]):-construction(C,closure,_,[DescID],Variables,_,_,_,_,_),
                               genericity(D,DescID,G2),
                               totypelist(Variables,TypeList).


genericity(C,C,G):-construction(C,composition,_,[X|_],_,_,_,_,_,_),
                   genericity(X,C,G).

genericity(D,C,G):-construction(C,composition,_,[X|_],_,_,_,_,_,_),
                   constituent(D,X),
                   genericity(X,X,G2),G2=[],
                   genericity(D,X,G).

genericity(D,C,G):-construction(C,composition,_,[X|_],_,_,_,_,_,_),
                   constituent(D,X),
                   genericity(D,X,[_|G]).

genericity(D,C,G):-construction(C,composition,_,[_|Y],_,_,_,_,_,_),
                   member(Yi,Y),
                   constituent(D,Yi),
                   genericity(D,Yi,G).

genericity(D,C,G):-construction(C,execution,_,[X],_,_,_,_,_,_),
                   genericity(D,X,G).
```

Nyní sestavíme algoritmus pro určení a uložení všech konstrukcí s extenzionálním výskytem pomocí předchozích dvou algoritmů.

Název: determineExtensional
E = extSupositionConstructions
Pro každou konstrukci C v E proveď
    R = kořenová konstrukce C
    G=genericity(C,R)
    Pokud je G negenerický kontext
        ulož extenzionální výskyt konstrukce C

Implementace je rozložena do dvou jednodušších metod a vypadá následovně.

```
confirmExtensional(ID):-rootID(ID,RootID),genericity(ID,RootID,G),G=[],!.

determineExtensional:-extSupositionConstructions(ID),
                          confirmExtensional(ID),
                          saveOccurrence(ID,'Extensional'),fail.
determineExtensional.
```

Kde *rootID(+ID,-IDKořene)* je predikát, který nám pro konstrukci s daným ID vrací ID kořenové konstrukce. Důležité je, že v případě empirického výrazu vrací až první podkonstrukci která následuje dvojici uzávěrů λwλt.

V našem příkladu se konstrukce 4, 5 a 6 se vyskytují extntenzionálním ne-generickém kontextu a tedy podle definice mají taktéž extenzionální výskyt.

Poslední úkol je již triviální – pro konstrukce, které stále nemají uložen hyperintenzionální nebo extenzionální, uložím kontext intenzionální.

Implementace v Prologu:

```
determineIntensional:-construction(ID,_,_,_,_,_,_,_,_,unknown),
                          saveOccurrence(ID,'Intensional'),fail.
determineIntensional.
```

Celý proces rozpoznávání kontextu je složen následujícím způsobem.

```
computeContext:-determineHyperintensional,
                  determineExtensional,
                  determineIntensional.
```

3.2.4 Výpis do souboru XML

Výstupem programu je soubor XML obsahující stromy konstrukcí včetně definovaných globálních proměnných a entit. XML soubor tvoří kořenový element *source*, který obsahuje elementy všech ostatních prvků TIL-Scriptu (konstrukce, proměnné, entity). Strukturu těchto elementů si nyní popíšeme. Konstrukce jsou reprezentovány XML stromy vytvořenými z elementů *construction* s následujícími atributy:

ID – ID konstrukce

type – typ konstruovaný danou konstrukcí

construction – vyjádření konstrukce v TIL - Scriptu

constructionType – typ konstrukce (Uzávěr, Trivializace ...)

occurrence – výskyt konstrukce

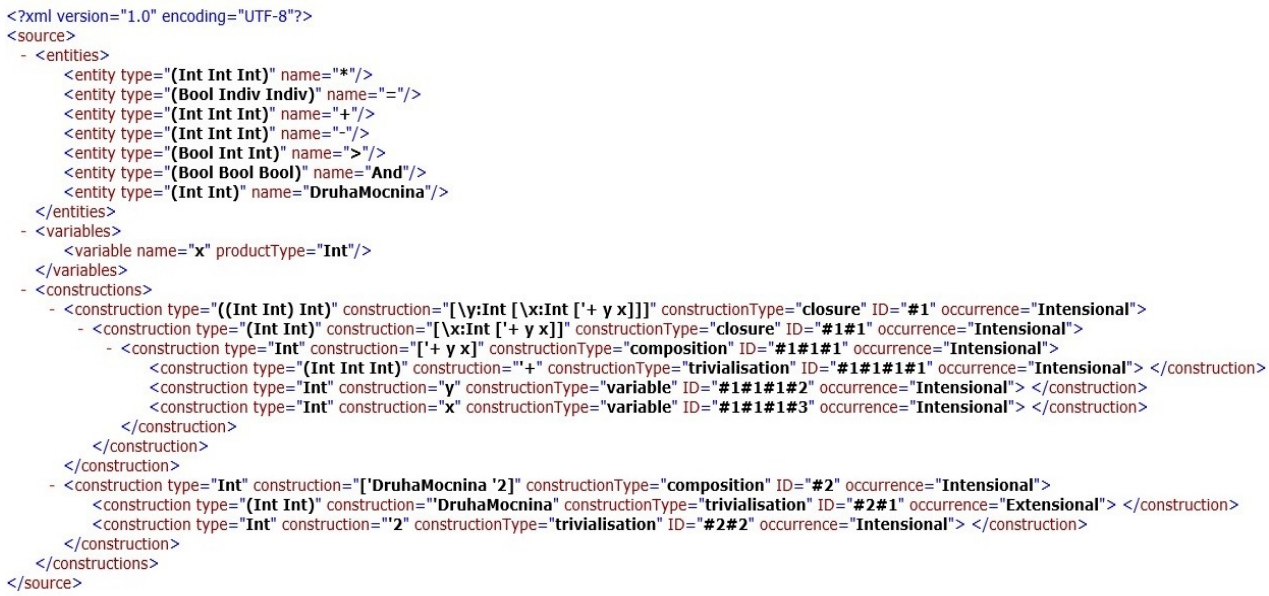
Jednotlivé entity jsou reprezentovány XML elementem *entity*. Jeho atributy jsou:
name – název entity
type – typ entity

U globálních proměnných se jedná o element *variable* s atributy:
name – název entity
productType – typ entity kterou proměnná konstruuje

Tyto informace jsou uzavřeny v elementech *entities* (resp. *variables*).

Důležité je před spuštěním výpisu do XML je potřeba zavolat predikát *tilTransfer/0.*Ten pro každou konstrukci vytvoří a uloží její vyjádření v TIL-Scriptu. Pro práci s XML souborem bylo potřeba nejprve sestavit několik pomocných predikátů, které budou s XML pracovat.

writeXMLheader – zapíše do souboru XML klasickou hlavičku, na rozdíl od následujících predikátů soubor vytvoří (případně přepíše), nikoliv zapisuje na konec existujícího
writeBeginMark(+Název) – zapíše do souboru počáteční značku XML elementu s daným názvem
writeEndMark(+Název) zapíše do souboru koncovou značku XML elementu s daným názvem
writeCon(+ID,+Tils,+TypKonstrukce,+KonstruovanýTyp,+Výskyt) – zapíše do souboru XML počáteční značku elementu cosntruction s uvedenými pěti parametry
writeEntites– zapíše do souboru XML všechny entity
writeVariables– zapíše do souboru XML všechny globální proměnné.



Obr. 3: ukázka výstupního XML souboru

Rekurzivní metoda *writeToXML(+ID)* pro výpis konstrukcí do souboru XML:

```
writeToXML(X):-construction(X,ConstructionType,_,DescIDs,
                  _,_,_,Tils,DataType,Occurence),
           writeCon(X,Tils,ConstructionType,DataType,Occurence),
           volej(DescIDs,writeToXML),
           writeEndMark(construction).
```

Metoda pro vytvoření celého XML souboru *writeToXML/0* vypadá takto.

```
writeToXML:-writeXMLheader,
       writeBeginMark(source),
           writeBeginMark(entities),
               writeEntities,
           writeEndMark(entities),
           writeBeginMark(variables),
               writeVariables,
           writeEndMark(variables),
           writeBeginMark(constructions),
               findall(ID,construction(ID,_,none,_,_,_,_,_,_,_),IDs),
               call2(IDs,writeToXML),
           writeEndMark(constructions),
       writeEndMark(source).
```

Volání pro každou kořenovou konstrukci probíhá nejdříve nalezením všech kořenových konstrukci, a následným voláním známého *call2/2*.

4. Závěr

V této práci jsem implementoval algoritmus rozpoznávání kontextu, ve kterém se může vyskytovat konstrukce jazyka TIL-Script, což je komputační varianta systému Transparentní Intensionální Logiky. K tomu, aby bylo dosaženo tohoto hlavního cíle práce, bylo nutno nastudovat poměrně bohatou teorii TIL, zejména pak systém definic, které určují způsob výskytu dané konstrukce v nadkonstrukci. Teprve pak bylo možno přistoupit k vlastní implementaci. Tato část se pak skládá z lexikální a syntaktické analýzy jazyka TIL-Script, která je prováděna v jazyce *C#*, a algoritmů pro rozpoznání kontextů a typovou kontrolu v Prologu. Během práce na těchto úkolech jsem si rozšířil znalosti o práci překladačů a o bezkontextových gramatikách. Taktéž jsem se naučil pracovat s Logickým programovacím jazykem Prolog.

Výsledný program zpracovává TIL-Script a rozpoznává kontexty správně ve všech testovaných případech. Do budoucna plánuji rozšíření jeho možnosti pracovat s objekty vyšších řádů, a přidáni lexikálního a syntaktického analyzátoru naimplementovaného taktéž v jazyce Prolog.

Výsledky, kterých bylo v této práci dosaženo, budou využívány v projektu GA15-13277S, „Hyperintensionální logika pro analýzu přirozeného jazyka“. Věřím, že budu v této práci pokračovat tak, aby mohl být algoritmus dále zdokonalován a postupně se tak stal základem pro inferenční stroj jazyka TIL-Script.

5. Reference

[1] Duží, M., Materna, P. (2012): *TIL jako procedurální logika: průvodce zvídavého čtenáře Transparentní intensionální logikou*. Bratislava: Aleph.

[2] Číhalová M., Duží M., Ciprich N., Menšík M. (2010): Agents' reasoning using TIL-Script and Prolog. *Frontiers in Artificial Intelligence and Applications*, Amsterdam: IOS Press, vol. 206, pp. 135-154.

[3] Ciprich N., Duží, M., Košinár M. (2009): The TIL-Script Language. *Frontiers in Artificial Intelligence and Applications,* Amsterdam: IOS Press, vol. 190, pp. 166-179.

[4] JANČAR, P. (2008): *Teoretická informatika* [online]. Ostrava: Vysoká škola báňská – Technická univerzita Ostrava.

[5] Duží, M. (2015): Procedurální teorie pojmů. Studia Philosophica 62, 2, 87-114.

[6] Duží, M., Jespersen, B. (2013*)*: Procedural isomorphism, analytic information, and β-conversion by value. *Logic Journal of the IGPL*, Oxford, vol. 21, pp. 291-308.

[7] Duží, M. (2014): Structural isomorphism of meaning and synonymy. *Computación y Sistemas*, vol. 18, No. 3, pp. 439–453.

[8] BRAMER, M. (2005): *Logic programming with Prolog*. New York: Springer.

[9] Duží, M. (2012): Towards an extensional calculus of hyperintensions. *Organon F*, vol. 19, supplementary issue 1, pp. 20-45.

[10] Duží, M. (2013): Deduction in TIL: From simple to ramified hierarchy of types. *Organon F*, vol. 20, supplementary issue 2, pp. 5-36.

[11] Duží, M. (2012): Extensional logic of hyperintensions. *Lecture Notes in Computer Science*, vol. 7260, pp. 268-290.

[12] Gordon, M.J.C., Melham, T.F. (1993): *Introduction to HOL.* Cambridge University Press.

Seznam příloh

Součástí diplomové práce jsou přílohy na CD.
CD obsahuje testovací soubor TIL-Skriptu a výsledný program jako projekt Visual Studia 2012 se zdrojovými kódy a spustitelnými soubory.

Příloha A: Gramatika TIL-Skriptu

```
start = {
    type definition, termination |
    entity definition, termination |
    construction, termination |
    object definition, termination |
    global variable definition, termination
};

termination = optional whitespace,".",optional whitespace;

type definition = "TypeDef", whitespace,
    type name, optional whitespace,
    ":=", optional whitespace,
    data type
;

object definition = "ObjectDef", whitespace,
    object name, optional whitespace,
    ":=", optional whitespace,
    construction
;
//type je datovy typ nebo definice typu(typedef ... )
entity definition = entity name,
       {optional whitespace,",", optional whitespace, entity name},
       optional whitespace, "/", optional whitespace,
       type
;

construction = trivialisation |
       variable |
       composition |
       closure |
       n-execution
;

global variable definition = variable name,
    {optional whitespace, ",", optional whitespace, variable name},
    optional whitespace, "->", optional whitespace,
    type
;

type= (data type | type name);

data type = "Bool"
    | "Indiv"
    | "Time"
    | "String"
    | "World"
    | "Real"
    | "Int"
    | "Any"
```

```
    | "List", optional whitespace, "(", optional whitespace, data type, optional
whitespace, ")"
    | "Tuple", optional whitespace, "(", optional whitespace, data type, optional
whitespace, ")"
    | "*"
    | "(", optional whitespace, data type1, optional whitespace ")"
    | "(", data type1, whitespace, data type1,")"
    | data type,'@tw'
;

data type1 = data type1, whitespace, data type1
    | data type


variable = variable name;

trivialisation = "'", optional whitespace, (construction | entity);

composition =
    "[", optional whitespace,
    construction,
    optional whitespace,
    construction,
    {construction},
    optional whitespace,
    "]"
    | construction, "@w",intnumber,"t",intnumber;
;

closure = "[",
    optional whitespace,
    lambda variables,
    optional whitespace,
    construction,
    optional whitespace,
    "]"
;

lambda variables = "\", optional whitespace, typed variables;

typed variables = variable name, optional whitespace, [":", optional
whitespace,data type],
    {optional whitespace,",", variable name, [optional whitespace, ":", optional
whitespace, data type]};

n-execution = "^", optional whitespace,nonzero digit, optional whitespace, (
construction | entity );

entity = keyword | entity name | number | symbol;

type name = upperletter name;

entity name = upperletter name;

variable name = lowerletter name;

object name = upperletter name;
```

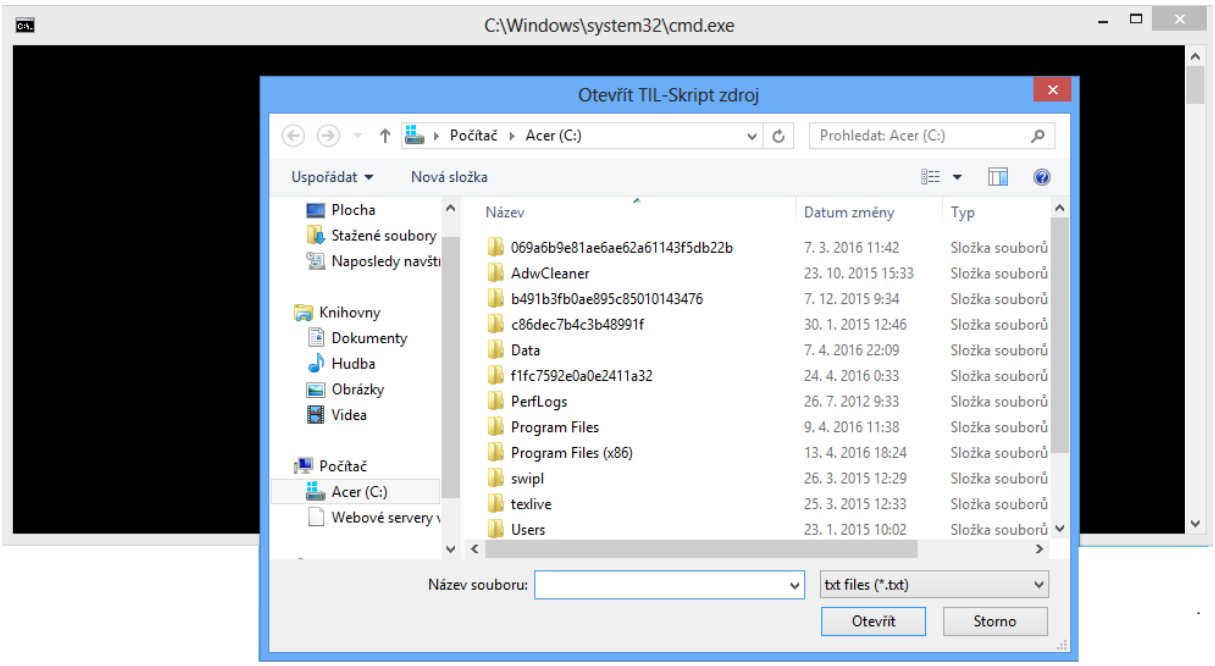
```
keyword =
    "ForAll" |
    "Exist" |
    "Every" |
    "Some" |
    "True" |
    "False" |
    "And" |
    "Or" |
    "Not" |
    "Implies"
;

lowercase letter = "a" | "b" | ... | "z";
uppercase letter = "A" | "B" | ... | "Z";

symbols = "+" | "-" | "*" | "/";

digit = "0" | nonzero digit;

nonzero digit = "1" | "2" | ... | "9";

number = intnumber[".", intnumber];

intnumber =  {digit};

upperletter name = uppercase letter, { lowercase letter | uppercase letter | "_" |
digit };

lowerletter name = lowercase letter, { lowercase letter | uppercase letter | "_" |
digit };

whitespace = whitespace character, optional whitespace;

optional whitespace = { whitespace character };

whitespace character = ? space ? | ? tab ? | ? newline ?;
```

Příloha B: Popis programu pro rozpoznávání kontextů.

Program na rozpoznání kontextu je jednoduchá konzolová aplikace.

Použiti programu:

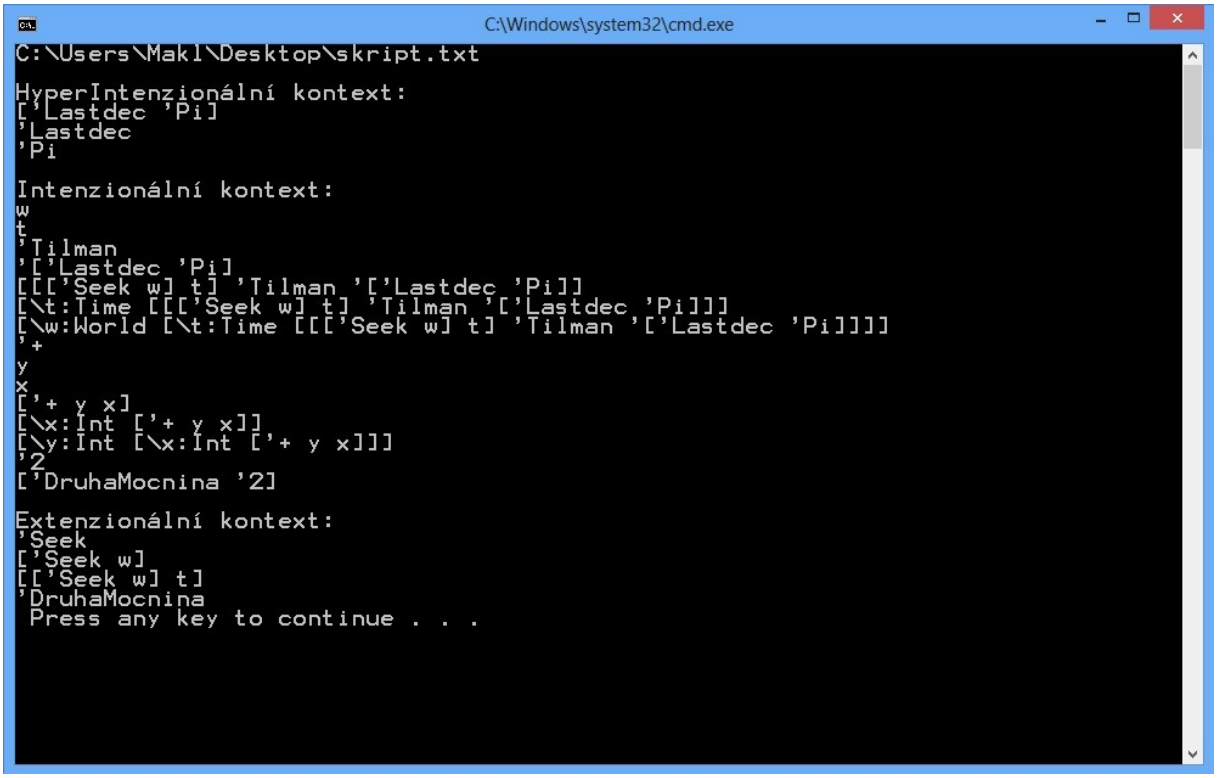
1) Po spuštění aplikace vyvolá dialog pro výběr souboru. Vstupním souborem je soubor TIL-Skriptu s příponou .tils nebo .txt.



1- Dialog pro výběr souboru

2) Po vybrání TIL-Skript souboru jej program zpracuje a vypíše výskyty. Tyto výskyty také uloží do souboru output.xml ve stejné složce, kde se nachází vstupní soubor. Pro ukázku je zvolen následující vstupní soubor:

```
x->Int .
Tilman/Indiv.
Seek / ( Bool Indiv *) @tw.
Lastdec/(Int Real).
Pi/Real.
[\w: World [\t:Time['Seek@wt 'Tilman '['Lastdec 'Pi]]]].
DruhaMocnina/(Int Int).
[\y:Int[\x:Int['+ y  x]]].
['DruhaMocnina '2].
```

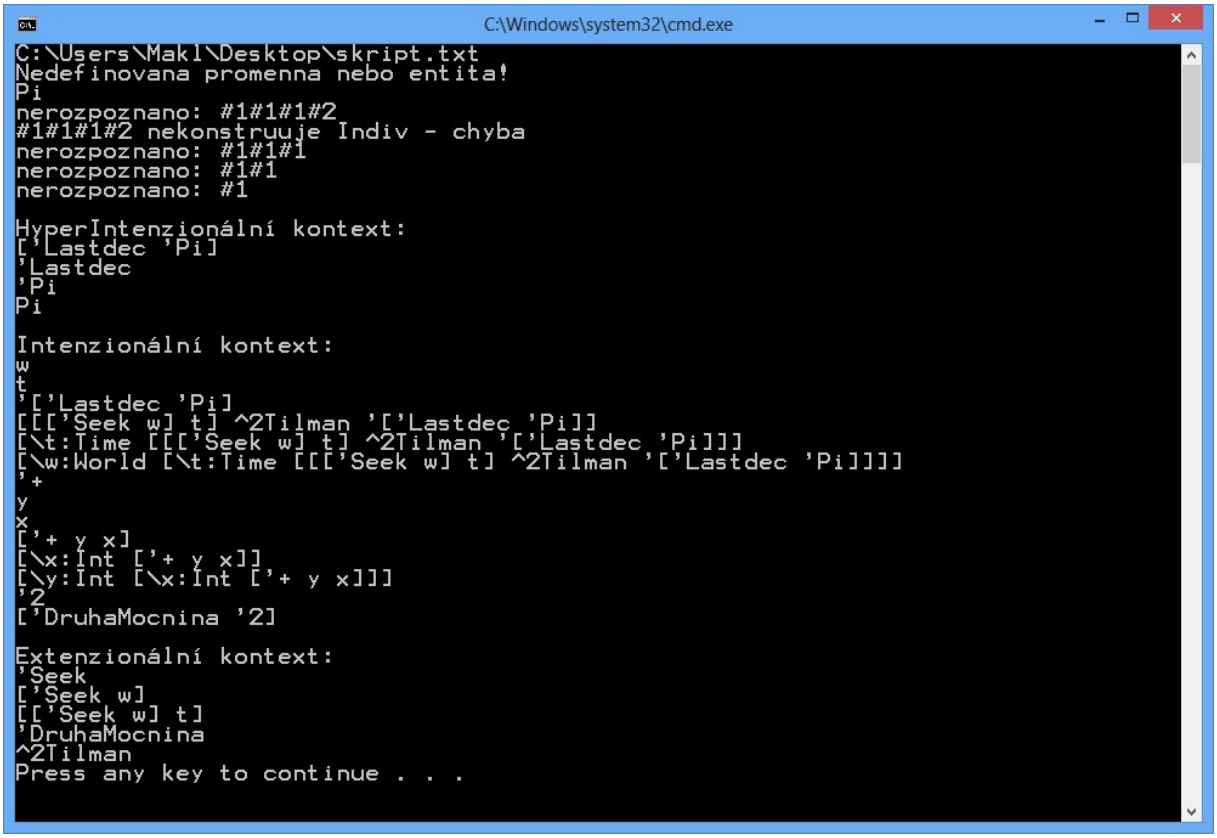


2 - výpis výsledků

Pokud dojde k chybě pří typové kontrole nebo program nenajde konkrétní definici entity či proměnné, program chyby vypíše, přesto však určí kontext (v následující ukázce jsme nahradili konstrukci `'Tilman` ve vstupu konstrukcí `^2Tilman` a odstranili definici čísla Pi). Na následujícím obrázku je ukázán výpis programu v případě chyb.

Nyní náš soubor vypadá následujícím způsobem:

```
x-> Int .
Tilman/Indiv.
Seek / ( Bool Indiv *) @tw.
Lastdec/(Int Real).
[\w: World [\t:Time['Seek@wt ^2Tilman '['Lastdec 'Pi]]]].
DruhaMocnina/(Int Int).
[\y:Int[\x:Int['+ y  x]]].
['DruhaMocnina '2].
```



3- chybové výpisy